Modeling of Hybrid Control Systems using the DEVSLib Modelica Library

Victorino Sanz[a,*], Alfonso Urquia[a], François E. Cellier[b], Sebastian Dormido[a]

[a] *Dpto. de Informática y Automática, ETSI Informática, UNED, Juan del Rosal 16, 28040, Madrid, Spain.*
[b] *Dept. of Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland.*

Abstract

DEVSLib is a free Modelica library, developed by the authors, that supports the Parallel DEVS formalism. The library is mainly designed to model discrete-event systems. It also includes interfaces to communicate the DEVSLib models with the rest of the Modelica libraries. Thus, the library can be used in the development of multi-domain and multi-formalism hybrid models using the object-oriented methodology supported by Modelica. This manuscript presents the hybrid system modeling capabilities included in DEVSLib. In particular, these functionalities are applied to the description of hybrid control systems. A case study of a supermarket refrigeration system, using the traditional control approach, is discussed. Three implementations of the plant and its controllers have been developed and are described. The system is simulated reproducing the day and night conditions, and the results from the three implementations are compared. DEVSLib is freely available

*Corresponding author. Tel: +34 913989469, Fax: +34 913987690
Email addresses: vsanz@dia.uned.es (Victorino Sanz), aurquia@dia.uned.es (Alfonso Urquia), francois.cellier@inf.ethz.ch (François E. Cellier), sdormido@dia.uned.es (Sebastian Dormido)

for download at `http://www.euclides.dia.uned.es`.

Keywords: Object-oriented modeling, hybrid control, Modelica, Parallel DEVS.

---

1. Introduction

Hybrid models, which define the interaction of continuous-time and discrete-event dynamics, are used for describing hybrid control systems, such as control systems where a continuous-time plant is controlled using discrete-time or event-based controllers.

Different modeling paradigms are applied for developing hybrid models. Two widely used paradigms are block diagram modeling and object-oriented modeling (i.e., physical modeling). The main advantage of the object-oriented modeling, in comparison with the block diagram modeling, is the acausal descriptions of models, which facilitates the modeling task and the code reuse (Cellier, 1991; Åström et al., 1998). Modeling environments supporting object-oriented modeling automatically perform the symbolic manipulations required to translate an acausal, object-oriented description of the model into efficient executable code (Cellier and Kofman, 2006).

Object-oriented modeling languages facilitate describing the continuous-time part of hybrid models using differential and algebraic equations (DAE). In addition, these languages provide constructs to describe discontinuities in the continuous-time behavior, equations with variable structure, and time and state events. Among the object-oriented modeling languages, the Modelica language (Elmqvist et al., 1998; Fritzson, 2003) may be considered the state-of-the-art for hybrid system modeling (Elmqvist et al., 1993; Mattsson

et al., 1999; Otter et al., 1999).

On the other hand, multiple formalisms have been proposed for representing discrete-event models. Finite Automata, Petri Nets, Grafcets and StateCharts are some of the most common formalisms used in control applications (Hrúz and Zhou, 2007). Extensions to these formalisms, such as Hybrid Automata (Lynch et al., 2003) and Hybrid Petri Nets (David and Alla, 2001), have been developed for modeling hybrid systems.

DEVS (Discrete EVent system Specification) is a modular and hierarchical formalism for modeling discrete-event systems (Zeigler et al., 2000). DEVS was proposed by Bernard P. Zeigler in 1976. Extensions to the DEVS formalism include Parallel DEVS (Chow, 1996), DEV&DESS for combined continuous-time and discrete-event systems (Zeigler et al., 2000), RT-DEVS for real-time discrete-event systems (Hong et al., 1997), Cell-DEVS for cellular automata (Wainer and Giambiasi, 2001), Fuzzy-DEVS (Kwon et al., 1996), and Dynamic Structure DEVS (Barros, 1995).

Some control applications of the DEVS formalism are described in (Zeigler, 1989). A DEVS-based methodology to analyze and design discrete-event controllers is proposed in Son and Kim (1994). Kofman (2003) presents the Quantized State Control (QSC) method, used to map an existing continuous-time controller into a discrete-event model within the DEVS formalism framework, which can be implemented in a digital computer. Also, Campbell and Wainer (2006) shows how to apply their M&S-Driven Engineering methodology, based on the use of the DEVS formalism, to develop components of real-time embedded systems.

The application of discrete-event modeling formalisms to the construction

of the discrete-event part of hybrid models facilitates the model development, maintenance, and reuse. It also helps to ensure the correctness and validity of the developed model.

Several Modelica libraries have been programmed in order to support discrete-event modeling formalisms. StateCharts (Ferreira and de Oliveira, 1999), StateGraph (Otter et al., 2005), Petri Nets (Mosterman et al., 1998) and Hybrid Automata (Pulecchi and Casella, 2008) are some of the supported formalisms. A more extensive list of available (free and commercial) Modelica libraries can be found at (Modelica Libraries, 2009).

DEVSLib is a free Modelica library that supports the Parallel DEVS formalism. The library includes interface models to transform DEVS events into discrete-time signals, and discrete-time and continuous-time signals into series of events. Thus, DEVSLib is compatible with other Modelica libraries, facilitating the description of multi-formalism and multi-domain hybrid systems. The DEVSLib library has been developed by the authors of this manuscript and is freely available on the web (www-euclides, 2009) as a package of the DESLib Modelica library (Sanz et al., 2009c). The library and the work discussed in this paper have been developed using the Dymola simulation environment (Dynasim AB, 2006).

The P-DEVS formalism has been chosen due to its versatility for hierarchical and modular description of discrete models. Object-oriented modeling languages have similar characteristics for describing continuous models. However, implementing the communication mechanisms of P-DEVS in Modelica turned out to be quite a challenge (Sanz et al., 2010).

It should be noted that the P-DEVS formalism can also be used as a basis

for implementing other discrete-event modeling approaches in Modelica, such as the process-oriented modeling approach (Kelton et al., 2007; Sanz et al., 2009c).

The objective of this manuscript is to present the functionalities provided by the DEVSLib Modelica library in order to describe hybrid control systems. The Parallel DEVS formalism and the Modelica language are combined to describe the controller and the plant. This combined approach facilitates the description of hybrid models, including the interactions between continuous and discrete parts.

DEVSLib has been successfully used to describe hybrid control systems. A hybrid model of a "Crane and Embedded Controller" system was described in Sanz et al. (2009a). This system was proposed by ARGESIM as a comparison for different tools that support hybrid modeling. The crane system was described as a continuous-time model using the Modelica language. The controller was modeled in part using the DEVSLib library and in part using the Modelica Standard Library (MSL). Both parts are interconnected using DEVSLib interface models. Another example of a hybrid opto-electrical communication system modeled using DEVSLib is described in Sanz et al. (2009b). DEVSLib has also been used to describe process-oriented models (Kelton et al., 2007) in Modelica (Sanz et al., 2006, 2007). These and other examples are distributed together with the library.

In this manuscript, a new case study is discussed to demonstrate the hybrid modeling functionalities of DEVSLib. The case study consists on a supermarket refrigeration system. This system was proposed in Larsen et al. (2007) as a benchmark for control applications. An NMPC control

approach for the system is discussed in Sarabia et al. (2009). The purpose of this work is to describe the system and its traditional control approach as a hybrid model using Modelica and DEVSLib following the object-oriented methodology.

The document is organized as follows: A brief description of the Parallel DEVS formalism and the Modelica language are included in Sections 2 and 3. The DEVSLib library is briefly described in Section 4, detailing its functionalities and interfaces for hybrid system modeling. The functionalities of DEVSLib applied to modeling of hybrid control systems are presented in Section 5. The supermarket refrigeration system modeled using DEVSLib and Modelica is discussed in Section 6. The manuscript ends with some conclusions.

2. Parallel DEVS Formalism

The Parallel DEVS (P-DEVS) formalism is briefly introduced in this section. Models in P-DEVS can be described behaviorally (named *atomic*) or structurally (named *coupled*).

2.1. Atomic P-DEVS Models

According to the P-DEVS formalism, an atomic model is the smallest component that can be used to describe the behavior of a system. An atomic P-DEVS model is formally defined by a tuple of eight elements, as described in (Chow, 1996):

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

where:

X	is the set of *input events*.
S	is the set of *sequential states*.
Y	is the set of *output events*.
$\delta_{int} : S \longrightarrow S$	is the *internal transition* function.
$\delta_{ext} : Q \times X^b \longrightarrow S$	is the *external transition* function.
$\delta_{con} : Q \times X^b \longrightarrow S$	is the *confluent transition* function.
$\lambda : S \longrightarrow Y^b$	is the *output* function.
$ta : S \longrightarrow \Re^+_{0,\infty}$	is the *time advance* function

X^b is a set of bags over elements in X, $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$, and e is the *time elapsed* since the last transition.

A detailed description of the behavior of an atomic P-DEVS model can be found in (Chow, 1996) and (Zeigler et al., 2000). An informal description of its behavior is presented here.

An atomic model remains in the state $s \in S$, for a duration $t_s = ta(s)$, if no input events are received during this interval. After t_s is elapsed, an *internal event* is triggered. The actions associated with the internal event are: 1) an output can be generated using the output function and the state previous to the event ($output = \lambda(s)$); and 2) an internal transition is performed, by changing the state to $s_{new1} = \delta_{int}(s)$.

Multiple input events can be received simultaneously through one or several ports:

– If any input event is received at time instant t_{ext} where $t_{ext} < t_{last} + t_s$, an *external event* is triggered. t_{last} indicates the time instant when the last event occurred, and so the input events are received before the

time instant for the next scheduled internal event. As a consequence of the external event, the state is changed to $s_{new2} = \delta_{ext}(s, e, bag)$ (i.e., an external transition is performed), where s is the current state, e is the elapsed time since the last transition ($t_{ext} - t_{last}$), and $bag \subseteq X$ is the set of received input events.

– If the external input event is received at time t_{ext} with $t_{ext} = t_{last} + t_s$, the conditions for the external and the internal events are simultaneously satisfied. This situation triggers a *confluent event* that substitutes the external and internal events. The actions associated with the confluent event are: 1) an output can be generated as $output = \lambda(s)$ (similarly to the management of internal events); and 2) a confluent transition is performed by changing the state to $s_{new3} = \delta_{con}(s, e, bag)$, being s the current state, e the elapsed time, and $bag \subseteq X$ the set of received input events (similarly to the δ_{ext} function).

A new internal event is scheduled after each internal, external, and confluent event. This new internal event will occur at time instant $t_{new} = ta(s_{new}) + time$, where $time$ is the current time, i.e., the time instant of the current event, and $ta(s_{new})$ is the duration until the next internal event scheduled as a consequence of the current event. The duration $ta(s_{new})$ is a function of the new state s_{new} (changed by the execution of the transition function). Note that the time advance function can also return a zero or an infinite value. If $ta(s_{new}) = \infty$, s_{new} is called a passive state in which the model will remain until an external input event is received. If $0 < ta(s_{new}) < \infty$, s_{new} is called an active state, and $ta(s_{new})$ indicates the time interval before the next internal event. If $ta(s_{new}) = 0$, s_{new} is called a transitory state,

which generates an immediate internal event.

2.2. Coupled P-DEVS Models

The P-DEVS formalism supports the hierarchical and modular description of the model. Every model has an interface to communicate with other models.

A coupled P-DEVS model is a model composed of several interconnected atomic or coupled models that communicate externally using the input and output ports of the coupled model interface. It is described by the following tuple (Zeigler et al., 2000):

$$M = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC)$$

where:

X	is the set of *input events*.
Y	is the set of *output events*.
D	is the set of the *component names*.
M_d	*DEVS model*, for each $d \in D$.
EIC	*External Input Coupling*: connections between the inputs of the coupled model and its internal components.
EOC	*External Output Coupling*: connections between the internal components and the outputs of the coupled model.
IC	*Internal Coupling*: connections between the internal components.

The connection of P-DEVS models implies the establishment of an information transmission mechanism between the connected models. P-DEVS models follow a message passing communication mechanism. A model generates messages as outputs, using its output function, which are received by other models as external inputs. Messages can be received simultaneously through one or multiple ports. Connections between models can be in the form of 1-to-1, 1-to-many and many-to-1. Each message can transport an arbitrarily complex amount of information (e.g., from one single number to a complex data structure, depending on the particular application). The received messages represent the bag of input elements used as an argument for the external and confluent transition functions.

3. The Modelica Language

Modelica is a free object-oriented modeling language mainly designed to describe mathematical models of physical systems (Modelica Association, 2009). Modelica is developed and maintained by the Modelica Association, which is an international association composed by multiple organizations and individual members. The development of the language includes several characteristics from previous languages like ALLAN (Jeandel et al., 1997), Dymola (Elmqvist, 1978), NMF (Sahlin et al., 1996), ObjectMath (Fritzson et al., 1995), Omola (Andersson, 1989), SIDOPS+ (Breuneuse and Broenink, 1997) and Smile (Kloas et al., 1995). Multiple free and commercial tools support the Modelica language such as CATIA (Dassault Systemes, 2009), Dymola (Dynasim AB, 2006), LMS Imagine.Lab AMESim (LMS International, 2009), MapleSim (Maplesoft, 2009), MathModelica (MathCore Engineering

AB, 2009), SimulationX (ITI GmbH, 2009), OpenModelica (Fritzson et al., 2002) and Scicos (Campbell et al., 2006).

Modelica supports the equation-based, object-oriented modeling methodology (EOO), which facilitates the description of large and complex systems (Cellier, 1996). Models can be described by means of equations and algorithms (i.e., behavioral description), by interconnecting instantiations of previously defined classes (i.e., structural description), or by combining both methods.

Modelica provides advanced modeling functionalities, including multiple class inheritance and definition of partial classes, constructs to impose information encapsulation, user's control over the selection of the state variables (Otter and Olsson, 2002), advanced features for model initialization (Mattsson et al., 2002), constructs to define arrays of components, connectors and connections, support to the combined use of equations, algorithms and structural information, and constructs for algorithmic Modelica code encapsulation and reuse, and for interfacing with external functions in C and Fortran (Olsson, 2005). The *replaceable* and *redeclare* constructs allow modifying the class of an object, even when already defined in a model. Models can also include annotations, which provide additional information such as graphical attributes, environment-dependent information, version and documentation.

Modelica supports event-based and non event-based treatment of logical conditions (using the *noEvent* operator). It provides language constructs to describe the trigger conditions of time and state events, and the actions associated to the events (Mattsson et al., 1999). These actions include: (1) updating the value of discrete-time variables and reinitialize continuous-time

state variables, using *when* clauses; and (2) changing the mathematical description of equations and assignments, using the *if* statement.

A model in Modelica has to satisfy the single-assignment rule. This means that the number of unknown variables and equations in the model has to be equal, and that the number of equations in each branch of a conditional equation must also be equal. Otherwise, the model is incorrect (*singular*).

Equations in Modelica follow the synchronous data flow principle, meaning that at each time instant, the active equations express relations between variables that have to be satisfied concurrently (Otter et al., 1999). The order in which the equations are evaluated is automatically determined by data flow analysis of the system of equations, leading to unique computations of the unknown variables.

There are different specialized classes to facilitate the description of diverse models. They include the *record*, *type*, *model*, *block*, *function*, *connector*, *package*, *operator*, and *operator function* classes. These classes present restrictions in the amount and type of components they may contain. A detailed description can be found in (Modelica Association, 2009).

Models can be described in a hierarchical and modular fashion, interconnecting components similar to the topological structure of the real system. Connections between EOO models are non-directional. Modelica provides the *connector* class, to describe the model interface, and the *connect* sentence, to describe the interactions (or connections) between models. These model interactions are based on the energy-balance principle. Connector variables are classified into *across* or *through*. The connected across variables are set equal, while the connected through variables are summed up

and the sum is set equal to zero.

The features for developing and using model libraries strengthen the Modelica modeling capabilities. Modelica libraries facilitate the application of several modeling formalisms and the development of multi-domain models (Modelica Libraries, 2009). The main Modelica library is the Modelica Standard Library (MSL) (Modelica, 2008), which is developed and supported by the Modelica Association.

Modelica modeling environments implement hybrid-model simulation algorithms (Elmqvist et al., 2001; Mattsson et al., 1999). Models are translated by the modeling environment into a hybrid DAE form, in order to simulate and analyze the system. The formal description of a hybrid DAE is (Modelica Association, 2009):

$$c := f_c(relation(v))$$

$$m := f_m(v, c)$$

$$0 = f_x(v, c)$$

with $v := [\dot{x}, x, y, t, m, pre(m), p]$, where:

- p are the variables without time dependency (i.e., parameters or constants).
- t is the independent variable (time).
- $x(t)$ is the set of variables that appear differentiated.
- $m(t_e)$ are the variables that are unknown and only change their values at event instants t_e. $pre(m)$ are the values of these variables immediately before the event.

– $y(t)$ is the set of algebraic variables.

– $c(t_e)$ are the conditions of all if-expressions, included when-expressions after conversion.

– $relation(v)$ are the relations containing variables v_i (e.g. $v_1 < v_2$).

This description defines a DAE that may include discontinuities, variable structure and/or discrete-events. The simulation is performed as follows (Modelica Association, 2009):

1. The DAE model is solved by a numerical integration method. In this phase, the conditions of the *if* and *when* clauses as well as the discrete variables are kept constant. Therefore, the model is a continuous function of continuous variables, and the most basic requirement of numerical integrators is satisfied.
2. During integration, all relations between model variables are monitored (e.g., $v_1 < v_2$). If one of the relations changes its value, an event is triggered, i.e., the exact time instant of the change is determined, and the integration is halted. Relations that depend only on time are usually treated in a special way, because this allows to determine the time instant of the next event in advance.
3. At an event instant, the model is a mixed set of algebraic equations that is solved for the Real, Boolean and Integer unknowns.
4. After an event is processed, if any of the relations between variables has changed its value due to the treatment of the event, a new event is triggered and processed. Otherwise, the integration is restarted (i.e., back to step 1).

4. The DEVSLib Modelica Library

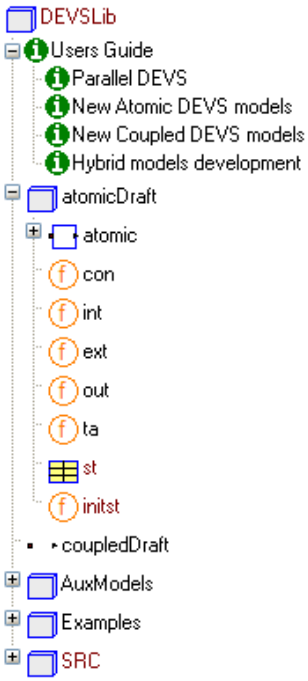


Figure 1: DEVSLib library architecture.

The general architecture of DEVSLib is shown in Fig. 1. Its components are structured in two areas: the user's area and the developer's area. The user's area contains all the required components to describe new P-DEVS models, including useful auxiliary models and the interfaces to combine DEVSLib models with other Modelica libraries. This area is composed of the Users Guide, atomicDraft, coupledDraft, AuxModels and Examples packages shown in Fig. 1. The developers area is contained in the SRC package (see Fig. 1), and includes the internal implementation of the library components as well as the development oriented documentation.

The behavior of a general P-DEVS atomic model is implemented in an abstract class called "AtomicDEVS", included in the SRC package. This class manages the detection of external, internal, and confluent events, and the execution of the actions associated with each transition.

Using the discrete event management functionalities included in Modelica, i.e., such as the *when* and *if* statements, the AtomicDEVS model defines the conditions for detecting the occurrence of external, internal, and confluent events. External events are detected due to the reception of an external message, which increases the value of the "event" variable of the input port that received the message. Internal events are detected when the simulation time reaches the time for the next scheduled internal event (i.e., $time >= nextInternalEvent$). Confluent events are detected when the conditions for external and internal events are simultaneously met. Dymola automatically detects the occurrence of these events during the simulation (Elmqvist et al., 2001; Mattsson et al., 1999).

The simulation time is managed by Dymola, which executes the numerical integrator and performs the simulation following the procedure described in Section 3. The simulation is performed combining the continuous-time integration, and the detection and execution of discrete events.

4.1. Atomic and Coupled Models with DEVSLib

Atomic models in DEVSLib (cf. model "atomicDraft" in Fig. 1) can be described analogous to their formal specification (see Section 2). The interfaces of the model are defined including the required input and output ports. The state is represented as the Modelica record "st" and can be initialized defining the "initst" function. The transition, output, and time

advance functions can be defined be means of the "int," "ext," "con," "out," and "ta" functions.

DEVSLib models can be aggregated and connected to compose coupled models. Coupled model components can be either atomic or other coupled models, thus the coupled models can be arranged hierarchically. The description of P-DEVS coupled models using DEVSLib follows their formal specification (see Section 2). Using the object-oriented modeling capabilities of Modelica, coupled models are constructed connecting previously developed components and including the required input/output ports.

4.2. DEVSLib Model Communication

In P-DEVS, external events transport information between models. The transmitted information is called "message". Messages are created by the output function and received as inputs for the external and confluent transition functions.

The model communication mechanism in Modelica is based on the definition of ports, called "connectors," and connections between ports, using "connect-equations". Variables defined in two connected connectors are either set equal, or are summed up with the sum being set equal to zero.

The Modelica model communication and the P-DEVS message passing mechanisms are conceptually different. The former relates values of variables, while the latter transports information between models. Several approaches were studied and developed in order to implement a suitable message passing mechanism in Modelica (Sanz et al., 2008).

A direct implementation of a message passing mechanism in DEVSLib using Modelica connectors was studied. However, connectors do not allow

the simultaneous reception of messages, because their variables cannot be assigned with several values at the same time. Also, Modelica does not allow a variable number of objects in a model, so the message transmission cannot be directly implemented.

Other approaches for developing the message passing mechanism in DEVSLib, based on an intermediate storage of the transmitted messages, were studied and implemented. The first approach was to use a text file to store the messages, so the sender writes the message to the file and notifies this to the receiver, that subsequently reads it. This approach allows simultaneous reception of messages, because several messages can be written to the file, but its performance and versatility are poor. The other approach substitutes the text files by dynamic memory space. This increases the performance and the versatility of the mechanism, allowing to manage different types of messages without redefining the message management operations.

The dynamic memory approach for message passing is the mechanism implemented in DEVSLib. This approach is combined with the standard Modelica connectors to provide a communication mechanism transparent to the user. DEVSLib models are topologically connected using standard Modelica connectors and connect-equations.

4.3. DEVSLib Interfaces to Other Modelica Libraries

DEVSLib includes models to translate the content of the DEVS messages into discrete-time signals, and discrete-time and continuous-time signals into DEVS messages. There are two mechanisms used in the continuous-to-discrete translation: the cross-functions and the quantization. The former translates the value of a continuous-time or discrete-time signal into a mes-

sage every time the signal crosses a given threshold in one direction (upwards or downwards). The models "crossUP" and "crossDOWN" implement this behavior in DEVSLib. The quantization mechanism is implemented by the "quantizer" model. This model generates a message every time the value of the continuous-time or discrete-time signal changes by a predefined quantum, similarly to the behavior of the Quantized State System (QSS) first-order integration method (Kofman, 2004).

On the other hand, the discrete-to-continuous translation is performed generating a piecewise-constant signal whose value is the one of the last message received. The model "DICO" (DIscrete-to-COntinuous) implements this behavior in DEVSLib. DEVSLib also includes a "DIBO" (DIscrete-to-BOolean) model that generates a boolean signal set to "false" if the value of the arrived message is zero, and "true" in any other case.

These interface models are implemented to manage the standard DEVSLib message type. However, the message type in DEVSLib can be redefined by the user and the interface models adapted to the new message type.

Also, DEVSLib models can receive continuous-time or discrete-time inputs as parameters to their transition functions. These inputs facilitate the description of discrete-event behavior in hybrid models, which can be influenced by the values of continuous-time or discrete-time variables. In order to maintain the modularity in the model construction, these inputs must be connected using the model interfaces.

5. Modeling of Hybrid Control Systems using DEVSLib

This section discusses the application of DEVSLib components and functionalities to describe hybrid control systems. The description of sensors and actuators and of discrete (time- or event-based) controllers is presented. The description of the plant is considered to be performed using a continuous-time model, and thus its description is not included in this section.

5.1. Sensors and Actuators

Sensors and actuators are required to communicate the continuous-time model of the plant with the controllers. Since time-based sensors are already available in the Modelica Standard Library, DEVSLib does not include them.

Event-based sensors can be modeled using the DEVSLib interfaces described in Section 4.3. The crossUP and crossDOWN models can be used as threshold detectors. The quantizer model can be used to observe the plant output avoiding time discretization (i.e., sampling).

The responses of these three models when applied to a sinusoid signal are shown in Fig. 2. The crossUP model detects the sinusoid signal crossing the value 2 in upwards direction and generates a message with that value. The crossDOWN model detects the sinusoid signal crossing the value 2 in downwards direction and generates a message with that value. The quantizer model generates a message every time the value of the signal changes by an amount exceeding the defined quantum – i.e., at values 1 and 2, and then again at 1 and 0, and so on. It should be noticed that the signal never crosses the values 3 and -3, and therefore, messages are not generated at those points.

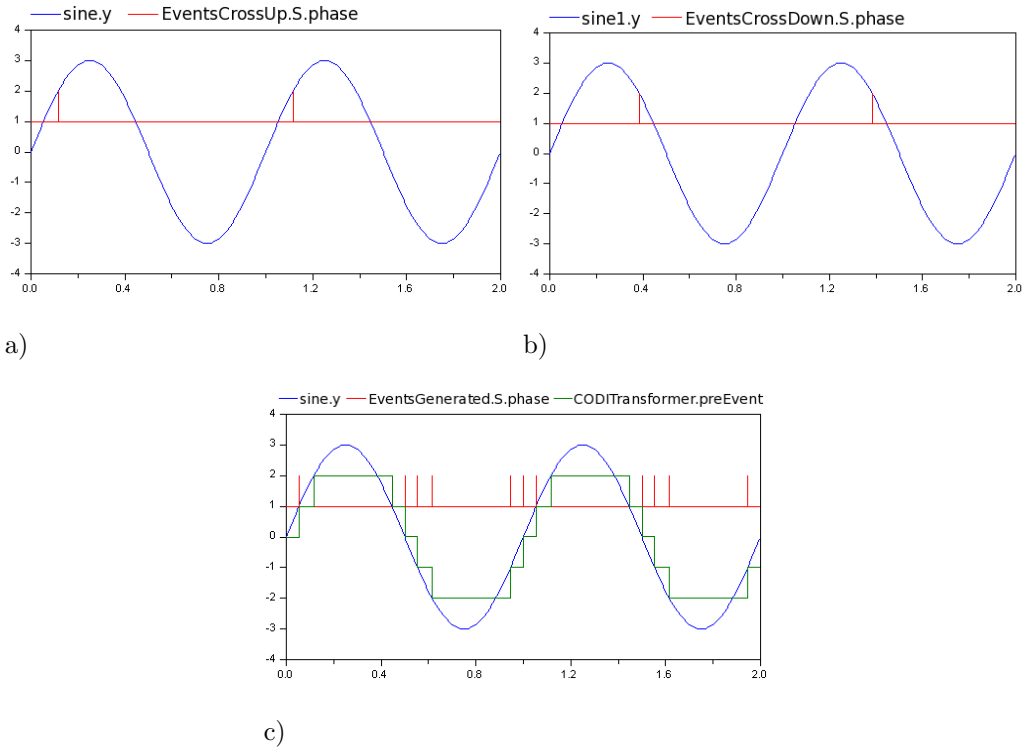


Figure 2: Sensors response using DEVSLib models: a) crossUP (value == 2); b) crossDOWN (value == 2); and c) quantizer (quantum == 1).

The output signal of an event-based controller is represented using messages. These messages have to be translated and communicated to the plant. The DEVSLib discrete-to-continuous models (i.e., DICO and DIBO) can be used to translate the generated control signal to a discrete-time or boolean signal. Also, DEVSLib includes the "setValue" model, which generates a message with a given value every time the model receives an external message. This model can be used to communicate control actions, independently of the value transmitted by the message.

For instance, consider the temperature control of a heating system. The

heater is turned off when the room temperature reaches a certain value. Using DEVSLib, this system can be implemented using a crossUP model to detect the temperature thershold. The output of the crossUP is connected to a setValue model, that sends a message with value 0. This message is received by a DIBO model that sets its output to false, due to the 0 value of the message, turning off the heating system. The description of this simple system using DEVSLib is shown in Fig. 3.

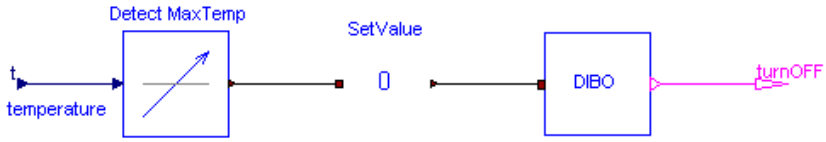


Figure 3: Simple temperature control system described using DEVSLib.

5.2. Controllers

DEVSLib can be used to describe discrete-time and event-based controllers. Depending on the complexity of the actions performed by the controller, it can be implemented as a single atomic model or as a coupled model. The transition functions of atomic models may contain the algorithms to calculate the control signal. On the other hand, coupled models can be constructed by combining simpler actions (e.g., the temperature control shown in Fig. 3).

Discrete-time controllers can be described using an atomic DEVSLib model that only executes periodic internal transitions. Each internal transition represents a sample interval. The time advance function schedules a new internal transition for the next sampling time. The inputs for the controller

are represented by continuous-time inputs for the atomic DEVSLib model, as described in Section 4.3.

Event-based controllers can be described using either atomic or coupled DEVSLib models, depending on their complexity. Controller inputs are received as messages through the input ports from the DEVSLib sensors, and the control signals are also generated as messages sent through the output ports.

6. Case Study: Supermarket Refrigeration System

The DEVSLib functionalities for describing hybrid control systems are applied to the design of a supermarket refrigeration system. This system was proposed in Larsen et al. (2007) as a benchmark for hybrid control applications.

A supermarket refrigeration system is composed of three main components: the display cases, the suction manifold and the compressor rack. The display cases contain the refrigerated goods offered to the customers. These display cases contain an evaporator that is connected to a pressure line. The objective is to control the temperature of the goods, which is approximated by the temperature of the air inside the display. Some external disturbances affect the temperature of the air in the display.

The refrigerant in each display case flows into the suction manifold. A compressor rack provides refrigerant to the displays by compressing the refrigerant in the suction manifold. Each display has an inlet valve to control the flow of refrigerant. Each compressor in the rack can be switched on and off, depending on the pressure in the line. The traditional control approach

for these kind of refrigeration systems includes two main controllers: the air temperature control associated with the display cases, and the pressure control integrated into the compressor rack.

This section includes a description of the plant and the traditional control approach described in Larsen et al. (2007) and Sarabia et al. (2009). These components have been modeled using plain Modelica code, the Modelica Standard Library, and DEVSLib. A comparison of the simulation results obtained by these three implementations is included.

6.1. Display Case

The dynamics of the display case are represented by four state variables: the temperature of the goods, T_{goods}, the temperature of the air, T_{air}, the temperature of the evaporator wall, T_{wall}, and the mass of liquefied refrigerant in the evaporator, M_{ref}. The inputs of the display are: the pressure in the suction line, P_{suc}, the state of the inlet valve (open/close), $valve$, and the disturbance, $Q_{airload}$.

The following three equations describe the energy balance between the goods, the air curtain, and the evaporator (Larsen et al., 2007).

$$\frac{dT_{goods}}{dt} = -\frac{Q_{goods-air}}{M_{goods} \cdot Cp_{goods}} \tag{1}$$

$$\frac{dT_{wall}}{dt} = \frac{Q_{air-wall} - Q_e}{M_{wall} \cdot Cp_{wall}} \tag{2}$$

$$\frac{dT_{air}}{dt} = \frac{Q_{goods-air} + Q_{airload} - Q_{air-wall}}{M_{air} \cdot Cp_{air}} \tag{3}$$

where $Q_{airload}$ is the external disturbance on the air curtain, and

$$Q_{goods-air} = UA_{goods-air} \cdot (T_{goods} - T_{air}) \tag{4}$$

$$Q_{air-wall} = UA_{air-wall} \cdot (T_{air} - T_{wall}) \tag{5}$$

$$Q_e = UA_{wall-ref}(M_{ref}) \cdot (T_{wall} - T_e) \tag{6}$$

UA is the overall heat transfer between media (defined with subscripts). M denotates the mass, Cp the heat capacity of the media, and T_e the evaporation temperature (approximated by Eq. (7), in absence of pressure drop in the suction line).

$$T_e = -4.3544 \cdot P_{suc}^2 + 29.2240 \cdot P_{suc} - 51.2005 \tag{7}$$

The heat transfer coefficient between the evaporator wall and the refrigerant is a function of the mass of liquefied refrigerant (cf. Eq. (6)), which is approximated by the following linear function:

$$UA_{wall-ref}(M_{ref}) = UA_{wall-ref,max} \frac{M_{ref}}{M_{ref,max}} \tag{8}$$

The accumulation of refrigerant in the evaporator is described by:

$$\frac{dM_{ref}}{dt} = \begin{cases} \frac{M_{ref,max} - M_{ref}}{\tau_{fill}} & \text{if } valve = 1 \\ \frac{Q_e}{\triangle H_{lg}} & \text{if } valve = 0, M_{ref} > 0 \\ 0 & \text{if } valve = 0, M_{ref} = 0 \end{cases} \tag{9}$$

where τ_{fill} is the filling time of the evaporator, $\triangle H_{lg}$ is the specific latent heat of the remaining liquefied refrigerant in the evaporator, which is approximated by Eq. (10).

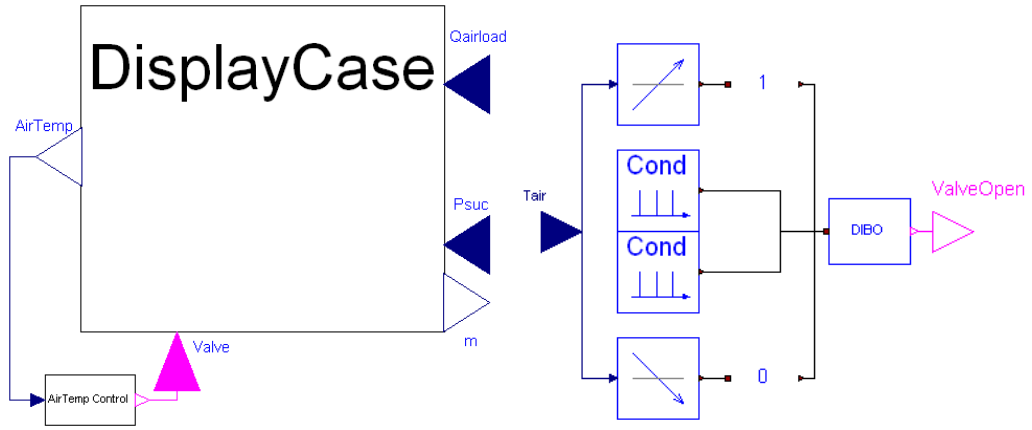
$$\triangle H_{lg} = (0.0217 \cdot P_{suc}^2 - 0.1704 \cdot P_{suc} + 2.2988) \cdot 10^5 \tag{10}$$

The mass of refrigerant leaving the evaporator into the suction manifold is described by:

$$m = \frac{Q_e}{\triangle H_{lg}} \tag{11}$$

The temperature control for the display case is defined as an hysteresis controller that opens and closes the inlet valves to regulate the temperature of the air, T_{air}. The parameters for the controller are the upper and lower thresholds for the temperature ($\overline{T}_{air}$ and $\underline{T}_{air}$). The hysteresis is operated at a sample time of 1 second. The state of the valves at the k^{th} sample is defined as:

$$valve(k) = \begin{cases} 1 & \text{if } T_{air} > \overline{T}_{air} \\ 0 & \text{if } T_{air} < \underline{T}_{air} \\ valve(k-1) & \text{if } \underline{T}_{air} < T_{air} < \overline{T}_{air} \end{cases} \quad (12)$$



a) b)

Figure 4: a) Display case, including air controller; and b) detail of air controller modeled using DEVSLib.

The model of the display case has been developed translating the equations described above into plain Modelica code. Interface ports have been added to the model in order to allow its connection with the other elements of the refrigeration system. The developed model is shown in Fig. 4.

The air controller detailed in Fig. 4b includes a crossUP model to detect the upper temperature for the air. When the air temperature reaches the threshold, the crossUP generates a message that is translated into another message with value 1 by the setValue model. This latter message is translated by the DIBO model into a "true" value for the valveOpen port. The crossDOWN model detects the lower limit for the air temperature and actuates analogously to the crossUP model, but in this case, the setValue generates a message with value 0, that closes the valve (i.e., sets the value of the valveOpen port to "false"). The two "Cond" models included in the center of the air controller check the initial conditions for the air temperature, setting the correct value for the valve at the beginning of the simulation.

6.2. Suction Manifold

The pressure of the suction line, P_{suc}, is described by:

$$\frac{dP_{suc}}{dt} = \frac{m_{in-suc} + m_{ref,const} - V_{comp} \cdot \rho_{suc}}{V_{suc} \cdot \frac{d\rho_{suc}}{dP_{suc}}} \tag{13}$$

where V_{suc} is the total volume of the suction manifold, V_{comp} is the volume flow created by the compressors, m_{in-suc} is the sum of refrigerant mass from the display cases into the suction manifold, $m_{ref,const}$ is a constant mass flow into the suction manifold from unmodeled entities, and ρ_{suc} is the density in the suction manifold (approximated by Eq. (14)).

$$\rho_{suc} = 4.6073 \cdot P_{suc} + 0.3798 \tag{14}$$

The model of the suction manifold has been developed similarly to the display case, translating Eq. (13) into plain Modelica code. The interface of the model is composed of three inputs (m, V_{comp}, and $m_{ref,const}$) and one output (P_{suc}).

6.3. Compressor Rack

The volume flow generated by each compressor is:

$$V_{comp,i} = comp_i \cdot \frac{1}{100} \cdot \eta_{vol} \cdot V_{sl} \quad i = 1, ..., q \tag{15}$$

where q is the number of compressors in the rack, $comp_i$ is the capacity of the i^{th} compressor ($\sum_{i=1}^{q} comp_i = 100$), η_{vol} is the volumetric efficiency, and V_{sl} is the total displacement volume.

The pressure control for the compressor rack is defined as a PI controller with a dead band (DB) around the reference pressure (cf. Eq. (16)). This controller is typically operated at a sample time of 60 seconds.

$$u_{PI}(t) = K_p e(t) + \int \frac{e(t)}{k_i} dt \tag{16}$$

where

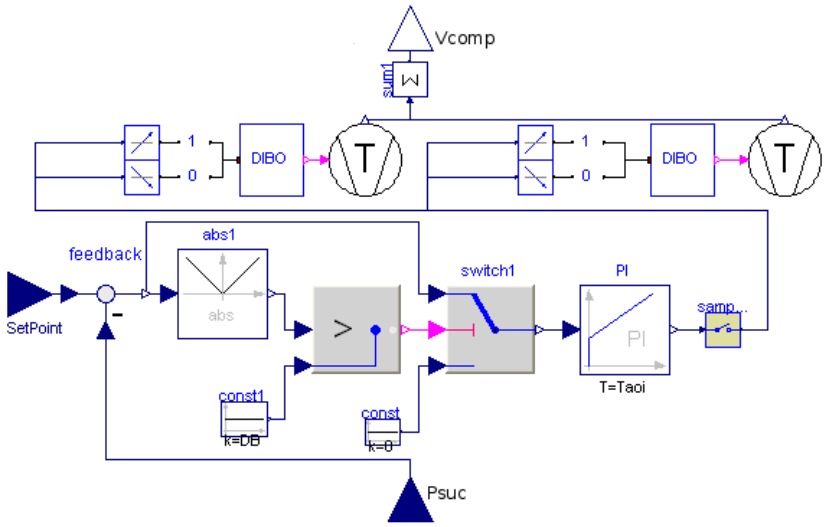
$$e(t) = \begin{cases} P_{suc}^{ref} - P_{suc} & \text{if } |e(t)| > DB \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

For all compressors in the rack, the nc^{th} compressor is switched on if Eq. (18) is satisfied, and switched off in all other cases.

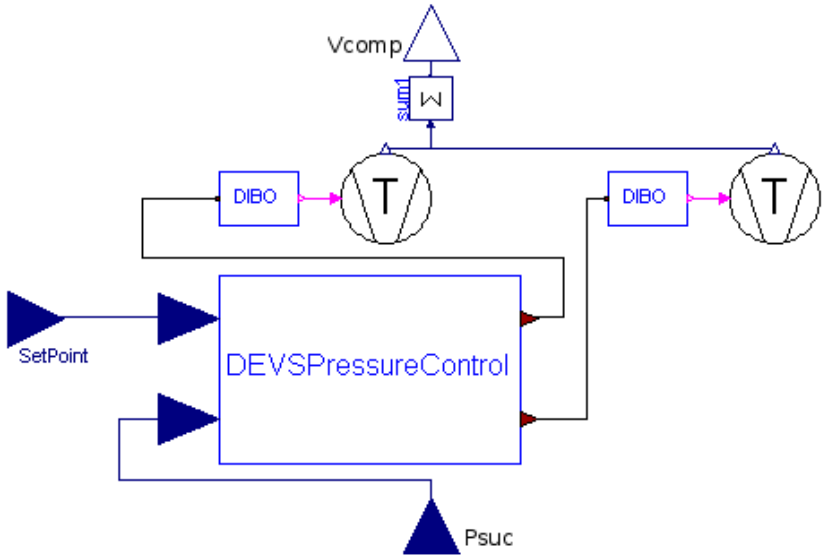
$$u_{PI} \geq \sum_{i=1}^{nc-1} comp_i + \frac{comp_{nc}}{2} \tag{18}$$

The compressor rack has been modeled using three different approaches. The dynamics of the compressors (Eq. (15)) have been modeled using plain Modelica code and are common for the three approaches:

– The first approach also uses plain Modelica code to describe the PI control.



a)



b)

Figure 5: Pressure control modeled using: a) DEVSLib and the MSL; and b) an atomic DEVSLib model.

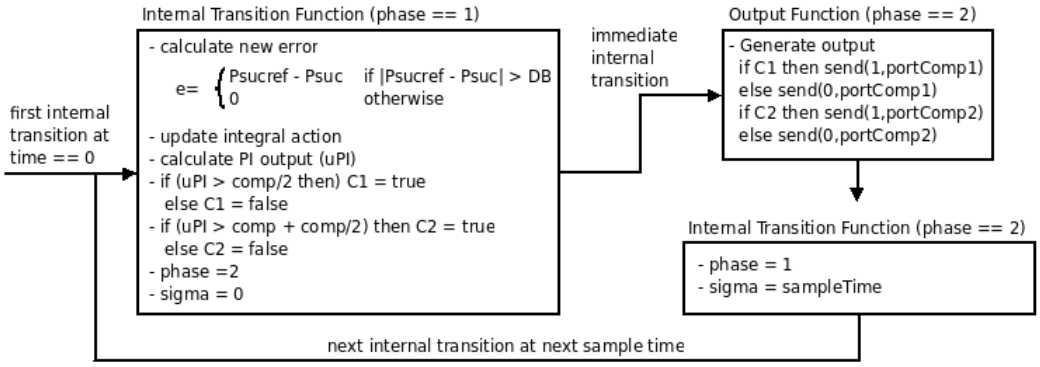


Figure 6: Actions performed by the atomic DEVSLib PI controller (note that no output is generated with phase == 1).

– The second approach uses elements from DEVSLib and the Modelica Standard Library to describe the PI control and the activation of the compressors (detailed in Fig. 5a). The sampled control signal generated by the PI controller is evaluated by "crossUP" and "crossDOWN" models in order to decide, which compressors have to be activated at each sample time.

– The third approach includes an atomic DEVSLib model that represents the PI control, and DIBO models to translate the generated control signal to the compressor models (detailed in Fig. 5b). In this case, the state of the PI controller is calculated at each sample time instead of calculating it continuously and only sampling its output. The actions performed by this atomic DEVSLib PI controller are shown in Fig. 6. The controller executes its first sampling at time 0s. At each sample time, it performs the following actions:

1. It executes the output function with phase == 1, and no output

is generated (not shown in Fig. 6).

2. It updates the state of the PI controller and decides the next state for the compressors.
3. It executes again the output function with phase == 2, and sends the new states to the compressors.
4. It executes again the internal transition function to schedule the next sample (sigma = sampleTime).

6.4. Experiment Setup and Simulation Results

The experiment performed with the whole refrigeration system is composed of two display cases, one suction manifold, and a compressor rack that includes two compressors. The developed model is shown in Fig. 7.

The system is evaluated during a day/night operation. During the day, the external disturbance in each display case is set to $3000J \cdot s^{-1}$. During the night, each display case is covered with "night-covers" that reduce the external disturbance to $1800J \cdot s^{-1}$ and the constant mass flow in the suction manifold from 0.2 to $0.0kg \cdot s^{-1}$.

The upper and lower temperature thresholds for the air in the display cases are set to $5°C$ and $2°C$, respectively. The reference pressure for the compressor rack is set to $1.4bar$ during the day, and $1.6bar$ during the night. Both compressors have the same capacity ($comp_1 = comp_2 = 50\%$).The rest of the parameters and initial conditions for the system are shown in Tables 1 and 2.

The system is simulated during $14.400s$, defining the switching between day and night at time $7.200s$. Simulation results are shown in Fig. 8, including the evolution of the air temperature values. The results obtained for the

Table 1: Parameters for the supermarket refrigeration system.

Display Cases		
M_{goods}	200	kg
Cp_{goods}	1000	$J \cdot kg^{-1} \cdot K^{-1}$
$UA_{goods-air}$	300	$J \cdot s^{-1} \cdot K^{-1}$
M_{wall}	260	kg
Cp_{wall}	385	$J \cdot kg^{-1} \cdot K^{-1}$
$UA_{air-wall}$	500	$J \cdot s^{-1} \cdot K^{-1}$
M_{air}	50	kg
Cp_{air}	1000	$J \cdot kg^{-1} \cdot K^{-1}$
$UA_{wall-ref,max}$	4000	$J \cdot s^{-1} \cdot K^{-1}$
$M_{ref,max}$	1	kg
τ_{fill}	40	s
Suction Manifold		
V_{suc}	5	m^3
Compressor Rack		
V_{sl}	0.08	$m^3 \cdot s^{-1}$
η_{vol}	0.81	$-$

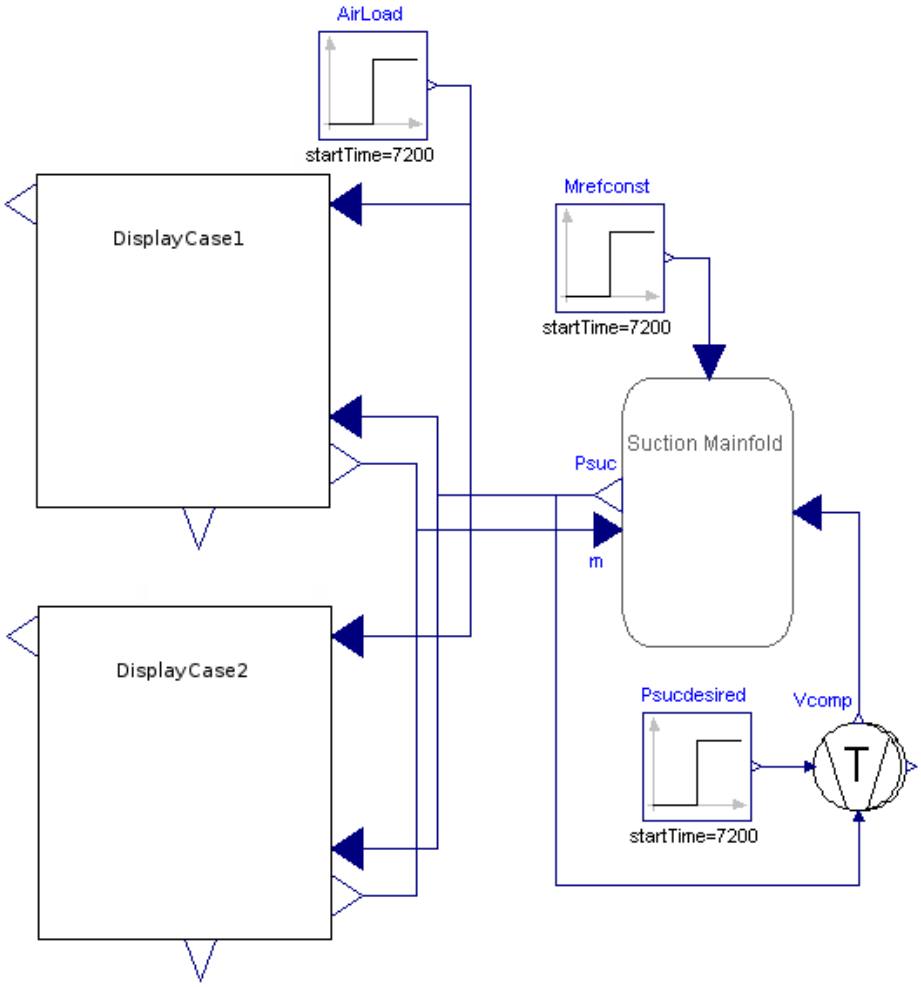
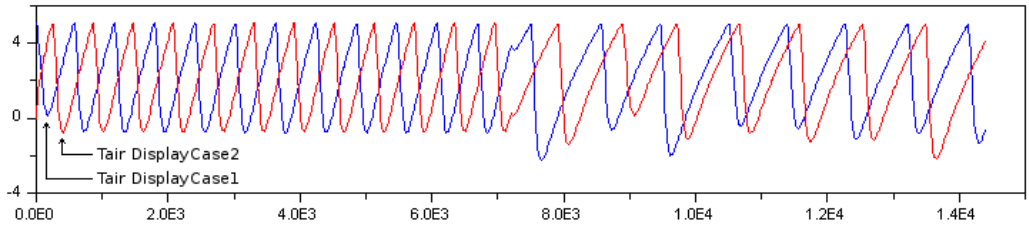
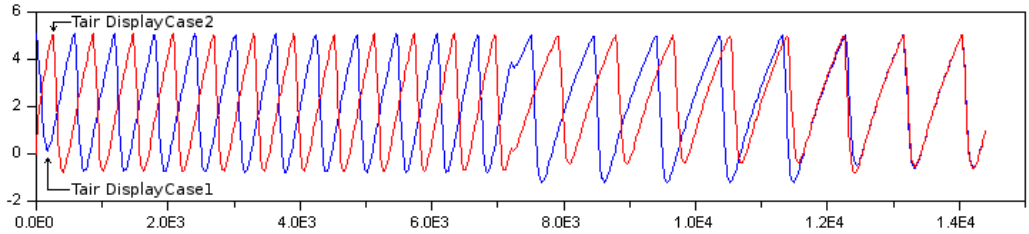


Figure 7: Supermarket refrigeration system modeled using DEVSLib.

models including the first and second pressure control approaches are identical (cf. Fig. 8a). The results obtained from the third approach (with the atomic DEVSLib pressure controller) are slightly different from the previous ones (cf. Fig. 8b). These differences are easily explained, because in the first and second approaches, the PI control operates in continuous-time and only



a)



b)

Figure 8: Evolution of air temperature values in both displays using: a) first and second control approaches; b) atomic DEVSLib control approach.

Table 2: Initial conditions for state variables.

	Disp. Case 1	Disp. Case 2
T_{wall}	0°C	0°C
T_{air}	5.1°C	0°C
T_{goods}	2°C	2°C
M_{ref}	0°C	0°C

its output is sampled, whereas in the atomic DEVSLib PI controller, all the calculations are performed at the same time and remain constant between samples. The results obtained with the third approach are similar to the results obtained by Sarabia et al. (2009), with a model of the supermarket

refrigeration system constructed using EcosimPro.

7. Conclusions

DEVSLib is a free Modelica library, developed by the authors of this manuscript, that supports the Parallel DEVS (P-DEVS) formalism. The DEVSLib functionalities and their application to the modeling of hybrid control systems have been discussed.

The descriptions of atomic and coupled P-DEVS models using DEVSLib are very close to their formal specifications. The model construction is facilitated by the object-oriented modeling capabilities provided by the Modelica language. DEVSLib includes interface models to combine P-DEVS models with the remainder of the Modelica libraries, which facilitates the development of multi-domain and multi-formalism hybrid models.

DEVSLib has been successfully applied to the description of a supermarket refrigeration system. An event-based controller for the air temperature of the display cases has been developed using DEVSLib. Also, two different controllers for the refrigerant pressure line have been developed. The first approach combines components from the Modelica Standard Library and DEVSLib. The second approach describes the pressure controller as an atomic DEVSLib model. The simulation results of the system using the first control approach are equivalent to the same controller developed using plain Modelica. The results from the second approach are slightly different due to the discrete-event nature of the whole controller.

Acknowledgments

This work has been supported by the Spanish CICYT under the DPI2007-61068 grant.

The authors also wish to express their gratitude to Prof. Cesar de Prada and Dr. Daniel Sarabia from the Universidad de Valladolid (Spain) for the information provided about their model of the supermarket refrigeration system developed using EcosimPro (Sarabia et al., 2009).

References

Andersson, M., 1989. Omola - an object-oriented language for model representation. Tech. rep., TFRT 7417, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Åström, K. J., Elmqvist, H., Mattsson, S. E., 1998. Evolution of continuous-time modeling and simulation. In: Procceedings of the 12^{th} European Simulation Multiconference. Manchester, UK, pp. 9–18.

Barros, F. J., 1995. Dynamic structure discrete event system specification: A new formalism for dynamic structure modeling and simulation. In: Proceedings of the 1995 Winter Simulation Conference. Arlington, VA, USA, pp. 781–785.

Breuneuse, A. P. J., Broenink, J. F., 1997. Modeling mechatronic systems using the SIDOPS+ language. Simulation Series 29 (1), 301–306.

Campbell, A. S., Wainer, G., 2006. Applying DEVS modeling for discrete

event multiple model control of a time varying plant. In: Proceedings of the 2006 Winter Simulation Conference. Monterey, CA, USA, pp. 823–831.

Campbell, S. L., Chancelier, J.-P., Nikoukhah, R., 2006. Modeling and simulation in Scilab\Scicos. Springer, New York, NY, USA.

Cellier, F. E., 1991. Continuous System Modeling. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Cellier, F. E., 1996. Object-oriented modeling: Means for dealing with system complexity. In: Procceedings of the 15^{th} Benelux Meeting on Systems and Control. Mierly, The Netherlands, pp. 53–64.

Cellier, F. E., Kofman, E., 2006. Continuous System Simulation. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Chow, A. C. H., 1996. Parallel DEVS: a parallel, hierarchical, modular modeling formalism and its distributed simulator. Transactions of the Society for Computer Simulation International 13 (2), 55–67.

Dassault Systemes, 2009. Computer aided three dimensional interactive application. http://www.catia.com/.

David, R., Alla, H., 2001. On hybrid Petri Nets. Discrete Event Dynamic Systems 11 (1-2), 9–40.

Dynasim AB, 2006. Dymola dynamic modeling laboratory user's manual. http://www.dymola.com/.

Elmqvist, H., 1978. A structured model language for large continuous systems. Ph.D. thesis, Department of Automatic Control, Lunk Institute of Technology, Lund, Sweden.

Elmqvist, H., Cellier, F. E., Otter, M., 1993. Object-oriented modeling of hybrid systems. In: In Eurosim Simulation Congress.

Elmqvist, H., Mattsson, S. E., Otter, M., 1998. Modelica – the new object-oriented modeling language. In: Proceedings of the 12^{th} European Simulation Multiconference. Manchester, UK, pp. 127–131.

Elmqvist, H., Mattsson, S. E., Otter, M., 2001. Object-oriented and hybrid modeling in modelica. Journal Europen des systmes automatiss 35 (1), 1 X.

Ferreira, J., de Oliveira, J. E., 1999. Modelling hybrid systems using statecharts and Modelica. In: Proceedings of the 7^{th} IEEE International Conference on Emerging Technologies and Factory Automation. pp. 1063–1069.

Fritzson, P., 2003. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley-IEEE Computer Society Pr.

Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H., Karstrm, A., 2002. The open source Modelica project. In: Proceedings of the 2^{nd} International Modelica Conference. Oberpfaffenhofen, Germany, pp. 297–306.

Fritzson, P., Viklund, L., Fritzson, D., Herber, J., 1995. High-level mathematical modelling and programming. IEEE Software 12 (4), 77–87.

Hong, J. S., Song, H.-S., Kim, T. G., Park, K. H., 1997. A real-time discrete event system specification formalism for seamless real-time software development. Discrete Event Dynamic Systems 7 (4), 355–375.

Hrúz, B., Zhou, M., 2007. Modeling and Control of Discrete-Event Dynamic Systems. Springer, London, UK.

ITI GmbH, 2009. SimulationX. http://www.simulationx.com/.

Jeandel, A., Boudaud., F., Larivire, E., 1997. ALLAN Simulation release 3.1 description. M.DGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, France.

Kelton, W. D., Sadowski, R. P., Sturrock, D. T., 2007. Simulation with Arena (4^{th} ed.). McGraw-Hill, Inc., New York, NY, USA.

Kloas, M., Friesen, V., Simons, M., 1995. Smile - a simulation environment for energy sytems. System Analysis Modelling Simulation 18–19, 503–506.

Kofman, E., 2003. Quantized-state control: A method for discrete event control of continuous systems. Latin America Applied Research 33 (4), 399–406.

Kofman, E., 2004. Discrete event simulation of hybrid systems. SIAM Journal on Scientific Computing 25 (5), 1771–1797.

Kwon, Y. W., Park, H. C., Jung, S. H., Kim, T. G., 1996. Fuzzy-DEVS formalism: concepts, realization and applications. In: Proceedings of AIS'96. pp. 227–234.

Larsen, L. F. S., Izadi-Zamanabadi, R., Wisniewski, R., 2007. Supermarket refrigeration system - benchmark for hybrid system control. In: Proceedings of the European Control Conference. Kos, Greece, pp. 113–120.

LMS International, 2009. Imagine.Lab AMESim. http://www.lmsintl.com/imagine-amesim-intro.

Lynch, N., Segala, R., Vaandrager, F., 2003. Hybrid I/O autamata. Information and Computation 180 (1), 103–157.

Maplesoft, 2009. MapleSim. http://www.maplesoft.com/products/maplesim/.

MathCore Engineering AB, 2009. MathModelica System Designer. http://www.mathcore.com/products/mathmodelica/.

Mattsson, S. E., Elmqvist, H., Otter, M., Olsson, H., 2002. Initialization of hybrid differential-algebraic equations in Modelica 2.0. In: Proceedings of the 2^{nd} International Modelica Conference. Oberpfaffenhofen, Germany, pp. 9–15.

Mattsson, S. E., Otter, M., Elmqvist, H., 1999. Modelica hybrid modeling and efficient simulation. In: Proceedings of the 38^{th} IEEE Conference on Decision and Control. Phoenix, AZ, USA, pp. 3502–3507.

Modelica, November 2008. Modelica standard library. http://www.modelica.org/libraries/Modelica.

Modelica Association, 2009. Modelica language specification 3.1. http://www.modelica.org/documents.

Modelica Libraries, 2009. Modelica free and comercial libraries. http://www.modelica.org/libraries.

Mosterman, P. J., Otter, M., Elmqvist, H., 1998. Modelling Petri Nets as local constraint equations for hybrid systems using Modelica. In: Proceedings of the Summer Computer Simulation Conference. Reno, NV, USA, pp. 314–319.

Olsson, H., 2005. External interface to Modelica in Dymola. In: Proceedings of the 4^{th} International Modelica Conference. Hamburg, Germany, pp. 603–611.

Otter, M., Årzén, K.-E., Dressler, I., 2005. StateGraph - a Modelica library for hierarchical state machines. In: Proceedings of the 4^{th} International Modelica Conference. Hamburg, Germany, pp. 569–578.

Otter, M., Elmqvist, H., Mattsson, S. E., 1999. Hybrid modeling in modelica based on the synchronous data flow principle. In: Proceedings of the 10^{th} IEEE International Symposium on Computer Aided Control System Design. Kohala Coast, HI, USA, pp. 151–157.

Otter, M., Olsson, H., 2002. New features in Modelica 2.0. In: Proceedings of the 2^{nd} International Modelica Conference. Oberpfaffenhofen, Germany, pp. 7–1 – 7–12.

Pulecchi, T., Casella, F., 2008. HyAuLib: modelling hybrid automata in Modelica. In: Proceedings of the 6^{th} International Modelica Conference. pp. 239–246.

Sahlin, P., Brign, A., Sowell, E. F., 1996. The neutral model format for building simulation (v. 3.02). Tech. rep., Dept. of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden.

Sanz, V., Cellier, F. E., Urquia, A., Dormido, S., 2009a. Modeling of the ARGESIM "crane and embedded controller" system using the DEVSLib Modelica library. In: Proceedings of the 3^{rd} IFAC Conference on Analysis and Design of Hybrid Systems. Zaragoza, Spain.

Sanz, V., Jafer, S., Wainer, G., Nicolescu, G., Urquia, A., Dormido, S., 2009b. Hybrid modeling of opto-electrical interfaces using DEVS and Modelica. In: Proceedings of the DEVS Integrative M&S Symposium, Spring Simulation Multiconference.

Sanz, V., Urquia, A., Dormido, S., 2006. ARENALib: A Modelica library for discrete-event system simulation. In: Proceedings of the 5^{th} International Modelica Conference. Vol. 2. pp. 539–548.

Sanz, V., Urquia, A., Dormido, S., 2007. DEVS specification and implementation of SIMAN blocks using Modelica language. In: Proceedings of the Winter Simulation Conference 2007. pp. 2374–2374.

Sanz, V., Urquia, A., Dormido, S., 2008. Introducing messages in Modelica for facilitating discrete-event system modeling. In: Proceedings of the 2^{nd} International Workshop on Equation-Based Object-Oriented Languages and Tools. Paphos, Cyprus, pp. 83–94.

Sanz, V., Urquia, A., Dormido, S., 2009c. Parallel DEVS and process-

oriented modeling in Modelica. In: Proceedings of the 7^{th} International Modelica Conference. Como, Italy, pp. 96–107.

Sanz, V., Urquia, A., Dormido, S., 2010. Integrating Parallel DEVS and equation-based object-oriented modeling. In: Proceedings of the DEVS Symposium, Spring Simulation Multiconference. Orlando, FL, USA.

Sarabia, D., Capraro, F., Larsen, L. F., de Prada, C., 2009. Hybrid NMPC of supermarket display cases. Control Engineering Practice 17 (4), 428–441.

Son, H. S., Kim, T. G., 1994. The DEVS framework for discrete event systems control. In: Proceedings of the AI, Simulation and Planning in High Autonomy Systems. Gainesville, FL, USA, pp. 228–234.

Wainer, G. A., Giambiasi, N., 2001. Timed Cell-DEVS: Modeling and simulation of cell spaces. In: Sarjoughian, H. S., Cellier, F. E. (Eds.), Discrete Event Modeling and Simulation Technologies: A Tapestry of Systems and AI-Based Theories and Methodologies. Springer.

www-euclides, 2009. Euclides web-site. http://www.euclides.dia.uned.es/.

Zeigler, B. P., 1989. DEVS representation of dynamical systems: Event-based intelligent control. Proceedings of the IEEE 77 (1), 72–80.

Zeigler, B. P., Kim, T. G., Praehofer, H., 2000. Theory of Modeling and Simulation. Academic Press, Inc., Orlando, FL, USA.

Figure Captions

Figure 1: DEVSLib library architecture.

Figure 2: Sensors response using DEVSLib models: a) crossUP (value == 2); b) crossDOWN (value == 2); and c) quantizer (quantum == 1).

Figure 3: Simple temperature control system described using DEVSLib.

Figure 4: a) Display case, including air controller; and b) detail of air controller modeled using DEVSLib.

Figure 5: Pressure control modeled using: a) DEVSLib and the MSL; and b) an atomic DEVSLib model.

Figure 6: Actions performed by the atomic DEVSLib PI controller (note that no output is generated with phase == 1).

Figure 7: Supermarket refrigeration system modeled using DEVSLib.

Figure 8: Evolution of air temperature values in both displays using: a) first and second control approaches; b) atomic DEVSLib control approach.