Simulation Modelling Practice and Theory 18 (2010) 998–1018

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

System modeling using the Parallel DEVS formalism and the Modelica language

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] Department of Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland

ARTICLE INFO

Article history:
Received 24 November 2009
Received in revised form 26 March 2010
Accepted 30 March 2010
Available online 4 April 2010

Keywords:
Parallel DEVS
Modelica
Discrete event
Hybrid systems modeling

ABSTRACT

The analysis and identification of the requirements needed to describe P-DEVS models using the Modelica language are discussed in this manuscript. A new free Modelica package, named DEVSLib, is presented. It facilitates the description of discrete-event models according to the Parallel DEVS formalism and provides components to interface with continuous-time models, which can be composed using other Modelica libraries. In addition, DEVSLib contains models implementing Quantized State System (QSS) integration methods. The model definition capabilities provided by DEVSLib are similar to the ones in the simulation environments specifically designed for supporting the DEVS formalism. The main additional advantage of DEVSLib is that it can be used together with other Modelica libraries in order to compose multi-domain and multi-formalism hybrid models. DEVSLib is included in the DESLib Modelica library, which is freely available for download at http://www.euclides.dia.uned.es.

1. Introduction

The Parallel DEVS (Discrete EVent Systems specification) formalism, first introduced by Chow and Zeigler [1], allows the modular and hierarchical specification of discrete-event systems.

Several simulation environments support the Parallel DEVS (P-DEVS) formalism, including DEVS-C++ [2], adevs [3], DEVSJAVA [4] and CD++ [5]. Some common characteristics of these environments are the following:

(1) As they are specifically designed for supporting the DEVS formalism, they do not facilitate the model description by combining different modeling formalisms. In particular, the continuous-time part of hybrid models has to be described applying DEVS-based techniques (e.g., integration algorithms based on state quantization techniques).
(2) The model is described using a programming language (i.e., C++ or Java).

On the other hand, the general-purpose, object-oriented modeling languages support the multi-formalism modeling of multi-domain hybrid systems. In particular, the Modelica language [6] facilitates the object-oriented description of hybrid systems. It supports a declarative description of the continuous-time part of the model (i.e., equation-oriented modeling) and provides language expressions for describing the occurrence of discrete-time events [7]. Models are mathematically described by differential, algebraic and discrete equations. These features have facilitated the development of Modelica

* Corresponding author. Tel.: +34 91 3089469.
E-mail addresses: vsanz@dia.uned.es (V. Sanz), aurquia@dia.uned.es (A. Urquia), francois.cellier@inf.ethz.ch (F.E. Cellier), sdormido@dia.uned.es (S. Dormido).

1569-190X/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2010.03.004

libraries [8] supporting several modeling formalisms (e.g., State Graphs [9], Petri nets [10] and bond graphs [11]) and describing phenomena in different domains (e.g., electrical, mechanical, thermo-hydraulic, chemical and process control). Also, model reusability is supported, reducing the costs and difficulty of new model development [12].

The Modelica language could be a vehicle for combining the use of the DEVS formalism with other modeling formalisms and techniques. The feasibility of describing atomic DEVS models in Modelica was demonstrated in [13]. Also, a Modelica library, called ModelicaDEVS [14,15], was developed for modeling continuous-time systems using the DEVS formalism and the Quantized State System (QSS) integration algorithms [16,17].

The analysis and identification of the requirements needed to describe P-DEVS models using the Modelica language are discussed in this manuscript. A new Modelica package intended to facilitate the application of the P-DEVS formalism is presented. This package, named DEVSLib, can be freely downloaded from [18], as a part of the DESLib Modelica library [19]. It facilitates the description of discrete-event models according to the P-DEVS formalism and provides components to interface with continuous-time models, which can be composed using other Modelica libraries. In addition, DEVSLib contains models implementing some of the QSS integration methods, which allow describing continuous-time models using DEVS-based techniques.

The description of an atomic DEVS model using DEVSLib is very close to its formal specification – i.e., it is performed by describing each element of the tuple. The transition, output and time-advance functions are specified using Modelica functions. This facilitates the model description and the understanding of the developed models. The description of coupled DEVS models with DEVSLib also matches completely with its formal specification. It is performed simply by connecting the corresponding ports of the component DEVS models.

In consequence, the model definition capabilities provided by DEVSLib are similar to the ones in the previously mentioned DEVS simulation environments, which are based on the use of programming languages such as C++ and Java. The main additional advantage of DEVSLib is that it can be used together with other Modelica libraries in order to compose multi-domain and multi-formalism hybrid models.

The manuscript has been structured in the following sections. Some fundamentals of the P-DEVS formalism and the Modelica language are briefly discussed in Sections 2 and 3. The requirements to describe P-DEVS models in equation-based object-oriented (EOO) modeling languages, and particularly in Modelica, are discussed in Section 4. An overview of the library is given in Section 5, presenting its general architecture of the library and main components. Sections 6–8 are devoted to discuss the functionalities included in the DEVSLib package in order to support the communication of P-DEVS models in Modelica, and the description of atomic and coupled P-DEVS models in Modelica. Sections 9–11, are devoted to discussing case studies the purpose of which is to illustrate the modeling capabilities of DEVSLib:

- The discrete-event model of an automatic teller machine described in [20] is employed to illustrate the development of atomic and coupled DEVS models using DEVSLib.
- The Lotka–Volterra model of predator–prey interactions [21,22] is described using the QSS methods for numerical integration supported by DEVSLib (i.e., QSS1, QSS2 and QSS3). The results and performance of the simulations are compared with the ones obtained using two different tools, PowerDEVS [23] and the ModelicaDEVS library [14,15].
- The tank system described in [24] is modeled using DEVSLib. This hybrid model illustrates the use of the DEVSLib interfaces between DEVS models and continuous-time models. The simulation results are compared with the ones obtained using the StateGraphs Modelica library [9].

The simulation of these case studies, and the development and validation of DEVSLib have been performed using Dymola [25].

2. Parallel DEVS formalism

The 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 a system. It is defined by a tuple of eight elements [26,27]:

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

where

$X_M = \{(p,v) \mid p \in IPorts,\ v \in X_p\}$	set of *input ports and values*
S	set of *sequential states*
$Y_M = \{(p,v) \mid p \in OPorts,\ v \in Y_p\}$	set of *output ports and values*
$\delta_{int} : S \to S$	*internal transition* function
$\delta_{ext} : Q \times X_M^b \to S$	*external transition* function, where $Q = \{(s,e) \mid s \in S, 0 \leqslant e \leqslant ta(s)\}$ is the *total state* set and e is the *time elapsed* since the last transition.
$\delta_{con} : Q \times X_M^b \to S$	*confluent transition* function.

$\lambda : S \rightarrow Y_M^b$	*output* function
$ta : S \rightarrow \Re_{0,\infty}^+$	*time-advance* function

An atomic model remains in the state $s \in S$, for a time interval $t_s = ta(s)$. After t_s is elapsed, an *internal event* is triggered and the state is changed to $s_{new} = \delta_{int}(s)$. Before that, an output can be generated using the output function and the state prior to the event ($output = \lambda(s)$).

A new internal event is scheduled to 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}.

Multiple inputs can be received simultaneously through one or several ports:

- If any input is received at time t_{ext} and $t_{ext} < t_s$ (so the inputs are received before the next internal event), an *external event* is triggered. As a consequence of the external event, the state is changed to $s_{new2} = \delta_{ext}(s, e, bag)$, where s is the current state, e is the elapsed time since the last transition ($t_{ext} - t_{last}$) and $bag \subseteq X_M$ is the set of received input messages.
- If the external input is received at time t_{ext} and $t_{ext} = t_s$, the external and the internal events are triggered simultaneously. This situation triggers a *confluent event* (that substitutes the external and internal events), and the state is changed to $s_{new3} = \delta_{con}(s, e, bag)$, being s the current state, e the elapsed time, and $bag \subseteq X_M$ the set of received inputs (similarly to the δ_{ext} function). Also, similarly to the internal events, an output can be generated as $output = \lambda(s)$ before executing the confluent transition function.

New internal events are also scheduled after the external and confluent transitions using $ta()$. Note that the time advance function can return a zero value, generating 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 [27]:

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

where

$X = \{(p,v) \mid p \in IPorts,\ v \in X_p\}$	set of *input ports and values*
$Y = \{(p,v) \mid p \in OPorts,\ v \in Y_p\}$	set of *output ports and values*
D	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 component
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, depending on the particular application or experiment being studied.

3. The Modelica language

Modelica [6] is a free modeling language mainly designed to describe mathematical models of physical systems. Modelica is developed and maintained by the Modelica Association. The development of the language includes several characteristics from previous languages like ALLAN [28], Dymola [29], NMF [30], ObjectMath [31], Omola [32], SIDOPS+ [33] and Smile [34]. Multiple free and commercial tools support the Modelica language such as CATIA [35], Dymola [25], LMS Imagine.Lab AMESim [36], MapleSim [37], MathModelica [38], SimulationX [39], OpenModelica [40] and Scicos [41].

Multiple Modelica libraries have been developed to facilitate the description of models using different formalisms and in multiple domains [8]. The possibility of reusing components from different libraries strengthens the Modelica modeling capabilities. The main Modelica library is the Modelica Standard Library (MSL) [42], which is developed and supported by the Modelica Association.

3.1. Characteristics of Modelica

A detailed description of the characteristics of the language can be found in the specification of the language [6]. Some of the characteristics of the Modelica language used for the development of the DEVSLib library are:

- *Description of models using acausal equations*. The causality is automatically assigned by the modeling environment by performing symbolic manipulations to the equations.
- *Combined use of equations and algorithms to define models*. The algorithms are executed imperatively, facilitating the description of behaviors with a fixed causality.
- *Reusable algorithm descriptions, as functions*. These allow to describe algorithmic operations as functions with parameters, and reuse them by simply calling the defined function using the appropriate parameters.
- *Information encapsulation*, that allows to hide information contained in a class that may not be relevant for outer classes or users. This functionality helps to structure the information contained in a model, and to avoid erroneous assignments or misuse of the internal components of a class.
- *Multiple class inheritance and definition of partial classes*, which include general properties of a class but cannot be instantiated. Classes may inherit information or characteristics from one or multiple classes, using the *extends* clause. This facilitates the description of common characteristics that are shared by several models or classes.
- *Class parametrization of the defined objects*. Using the *replaceable* and *redeclare* constructs it is possible to modify the class of an object, even when already defined in a model. It simplifies the experimentation with the model. The modeler is allowed to modify the class of a defined object instead of having to re-describe the model and its components.
- Provides language constructs to describe the trigger conditions of time and state events, and also the actions associated to the events [43,7,44]. These actions can be: (1) update the value of discrete-time variables; (2) reinitialize continuous-time state variables, using *when* clauses; and (3) change the mathematical description of equations and assignments, using the *if* statement.
- *Model annotations*, that may contain additional information of the model (i.e., the graphical representation, icon representation, environment-dependent information, version, documentation, etc.).
- *External function interface with C and Fortran*, which facilitates the inclusion of C and Fortran code into Modelica, extending the functionalities of Modelica with those of these general programming languages.

A model in Modelica may include the following components: (1) *parameters/constants* that represent entities the values of which remain constant during the simulation; (2) *variables* that represent entities with values that may vary during the simulation; (3) *algorithm sections* to describe algorithmic behavior (i.e., imperatively described and sequentially executed); (4) *equation sections* for the description of the relations between the variables of the model (algebraic and differential variables); and (5) *initial algorithms/equations* used to initialize the state of the model.

A model in Modelica has to comply with 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.

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 [44]. The set of active equations can be composed of: only continuous equations, during continuous integration, or mixed continuous and discrete equations, if an event has been triggered and needs to be evaluated. 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 [45]. Both, continuous and discrete, equations have to be considered during the sorting procedure in order to obtain the correct evaluation order for the possible sets of active equations.

The connections between models in EOO languages are based on the energy-balance principle. Modelica provides the *connector* class, to describe the model interface, and the *connect* sentence, to describe the interactions (or connections) between models. Variables in the connectors can be either *across* or *through*. Variables in Modelica connectors are described by default as across, and the *flow* modifier is provided to describe through variables. Across variables in a node (i.e., a connection point) assume the same value, while the through values are summed up and the sum is set equal to zero.

3.2. Simulation of Modelica models

Models in Modelica are described following the EOO modeling methodology. They are later translated by the modeling environment into a hybrid DAE form, in order to simulate and analyze the system. Hybrid DAE models may include discontinuities, variable structure and/or discrete-events [6].

The simulation is performed as follows [6]: (1) the continuous-time part is solved using a numerical integration algorithm; (2) if any of the event conditions is met during integration, the integration algorithm is halted and the event instant is determined; (3) at the event instant the set of algebraic and discrete equations are solved; and (4) once the event has been treated, the event conditions are checked again. If a new event is triggered, it is immediately executed (i.e., event iteration). Otherwise, the integration is restarted.

4. Integrating the P-DEVS formalism into EOO languages

In this section, the requirements needed to describe P-DEVS models using an EOO modeling approach are discussed. These requirements meet the necessity to describe atomic and coupled P-DEVS models, and the possibility to combine discrete-event and continuous-time models. The description of these requirements is particularly applied to the case of the Modelica language. The identification and analysis of the additional Modelica functionalities required to describe models following the P-DEVS formalism constitute the foundations of the work presented.

4.1. Discrete-event model behavior

P-DEVS models, as discrete-event models, have a fixed causality. The actions associated with the events are described algorithmically using functions. Discrete-time and event management constructs are required to describe the behavior of a P-DEVS model in EOO languages. The discrete part of the model can be described in different ways, depending on the functionalities provided by the language itself (i.e., algorithm sections [46], concurrent programming language statements [47], operating procedures [48] or event-driven processes [49,50]).

In general, EOO languages provide functionalities to manage discrete events. These functionalities have to be combined to reproduce the semantics of P-DEVS models (i.e., event detection, management and execution of transition functions), in order to facilitate the description of P-DEVS models in EOO languages.

Modelica provides language constructs to describe the trigger conditions of time and state events, and also the actions associated to the events [7]. These functionalities have been previously used to describe models following multiple formalisms, like State Charts, Petri Nets, State Graphs and Classic DEVS. The same functionalities can be used to describe the behavior of P-DEVS models. To this end, the detection of internal, external and confluent events has to be defined. Also, the actions associated with each type of event have to be managed (i.e., the execution of transition functions).

4.2. Model communication mechanism

Each P-DEVS model, atomic or coupled, has an interface to communicate with other models. These interfaces allow the composition of modular and hierarchical models, in order to construct more complex models. EOO models also contain model interfaces that allow the connection of multiple components in a similar fashion, to construct more complex models. However, the concepts underneath both model interfaces and their connections are different.

As previously described, model communication in P-DEVS follows a message passing mechanism. On the other hand, the connections between models in EOO languages are based on the energy-balance principle, establishing relationships between across and through variables. However, these language constructs are not enough to describe the required P-DEVS message communication mechanism, because:

- They do not allow the simultaneous transmission of messages from one port to another, due to the single-assignment rule.
- They do not allow to connect multiple output ports to the same model and transmit simultaneous messages, also due to the single-assignment rule.
- The amount of information transmitted by the connector is fixed by the number of variables in it.
- The structure of the information transmitted with the connection is also fixed due to the variables defined in the connector.

In order to allow the description of P-DEVS models in Modelica, a message passing mechanism has to be implemented. Ideally, this message passing mechanism should be transparent to the user, in order to facilitate the integration of both formalisms without increasing the complexity of model development.

4.3. Interfacing P-DEVS and other modeling formalisms

The idea is to combine models described using P-DEVS with models defined using other formalisms for continuous-time modeling (i.e., the physical modeling paradigm), using EOO languages. This combination facilitates the description of multi-formalism hybrid systems.

Two approaches for communicating P-DEVS models with other formalisms are proposed:

- *Translated Interface Connections*: connecting the output of a P-DEVS model to the input of a continuous-time model, or vice versa. Due to the mentioned differences in the model communication mechanism, it is required to define interface models that translate messages into discrete-time signals, and both continuous-time and discrete-time signals into messages. These interface models allow to couple discrete-event and continuous-time components together in the hierarchy of models that compose a hybrid system.
- *Direct Interface Connections*: allowing to describe the behavior of a discrete-event model which is influenced by the state of a continuous-time model. P-DEVS models could receive continuous-time or discrete-time signals as inputs to

its transition functions. In order to maintain the modularity in the model construction, these inputs must be connected using the model interfaces. These connections are similar to the interactions described in the DEV&DESS formalism between the discrete-event and the continuous-time parts of a hybrid model [27].

To combine P-DEVS models with models from other Modelica libraries, two types of model communication have to be supported:

- Interface models have to be constructed to translate messages, as described above, into discrete-time signals. Also, continuous-time and discrete-time signals from the Modelica models have to be translated into messages.
- The direct connections from Modelica to P-DEVS models can be supported by allowing continuous-time inputs for the transition functions. The value of the continuous-time signal connected to one of these inputs is used as an input for the transition function.

5. DEVSLib architecture

In order to facilitate the understanding and use of DEVSLib, it can be considered that its models are classified into two groups: the "user's area" and the "developer's area". The top level of the hierarchy is shown in Fig. 1a. The "user's area" consists of the *User's Guide*, the *atomicDraft* package, the *coupledDraft* model, the *AuxModels* package and the examples provided within the *Examples* package. The "developer's area" consists of a single package, the *SRC* package.

5.1. User's area

The "user's area" contains all models intended to be used directly by the library user, in particular, those needed to develop atomic and coupled P-DEVS models, to interface with continuous-time models and also the models implementing the QSS integration methods. The documentation of these packages addresses those users who wish to use the library but do not need to understand its internal design and implementation.

The structure of the "user's area" is shown with more detail in Fig. 1b. The *atomicDraft* package and the *coupledDraft* model are used to define new atomic and coupled P-DEVS models. Both will be detailed in Sections 7 and 8. The *AuxModels* package contains some useful auxiliary models that are usually needed. It includes the following models:

- Generator and Display are models that can be used as source and sink of messages, respectively.
- DUP and DUP3 are models that duplicate each incoming message and instantaneously send a copy of it through all its output ports (two in the case of DUP, and three in DUP3). The use of these models is detailed in Section 6.
- Select is a model that sends each received message through one of its two output ports, depending on a given boolean condition.
- BreakLoop is used to break algebraic loops in coupled models. Its use is detailed in Section 8.
- DiCO, DIBO, Quantizer, CrossUP and CrossDOWN are the interface models used to combine DEVSLib models with models from other Modelica libraries. Their use is detailed in Section 11.
- QSS1, QSS2 and QSS3 are models that implement the first, second and third order QSS integration methods. They are detailed in Section 10.

The *Examples* package contains several models that can help the user to learn and understand the use of the library. The models included are:

- SimpleModels includes simple atomic and coupled DEVSLib models. The implementations of the Generator and Display are included, as well as a Processor, Switch, Pipe, and other examples described in [27].
- ATM includes the model of an Automatic Teller Machine. The specification of this model can be found in [20], and its implementation using DEVSLib is detailed in Section 9.
- Clock2 includes the model of a pendulum clock. It is modeled as a hybrid system, with the pendulum represented by a continuous-time model and the rest of the clock by a P-DEVS model. The specification of the model can be found in [51].
- CarFactory includes a model of a simple car production factory [52].
- HybridONoC includes a hybrid model of an optoelectrical communication system. Detailed information about the model can be found in [53].
- QSSIntegration includes a differential equation, the Lotka–Volterra (detailed in Section 10) and a flyback-converter model implemented using QSS integration methods. Other required models such as adder, multiplier, gain, square-root, step, constant, and switch, are also included.
- ControlledTanks includes the model of a two-tank hybrid system with discrete controller. This system is detailed in Section 11.
- PetriNetsExamples includes the model of an MM1 queue system, in order to compare it with its implementation included in the Extended PetriNet Modelica library.

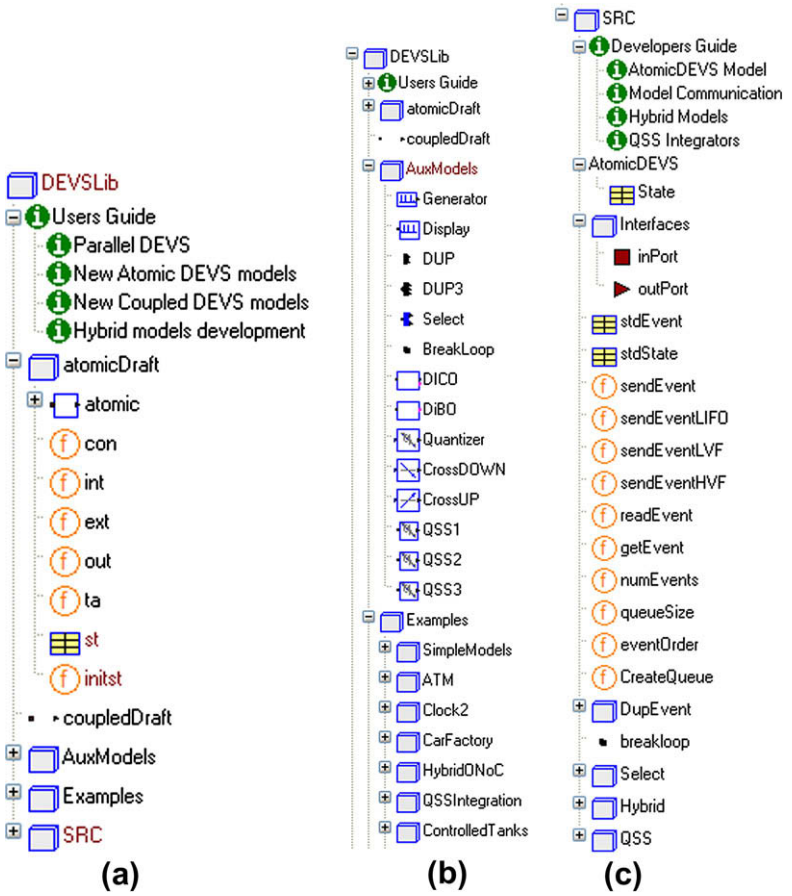

Fig. 1. DEVSLib library architecture: (a) general architecture; (b) user's area; and (c) developer's area.

5.2. Developer's area

In contrast, the "developer's area" contains data structures and partial models that the library user does not need to use directly. The documentation of this area addresses library developers.

The "developer's area" is shown in Fig. 1c. The *AtomicDEVS* model contains the Modelica implementation of the general behavior of an atomic P-DEVS model. This model is inherited by the *atomicDraft* package of the "user's area". In the AtomicDEVS model, a data structure (i.e., a record) represents the model state and Modelica functions describe the P-DEVS functions (i.e., state-initialization, transition, output and time advance functions).

In addition, the "developer's area" contains the implementation of input and output ports, functions supporting the message passing mechanism needed to communicate the DEVS models, the implementation of the event-duplicator model (DUP), the model to break algebraic loops (BreakLoop), the implementation of the Select model, the interfaces to combine DEVSLib with other libraries, and the QSS integration methods.

6. DEVSLib model communication

This section discusses the description of the communication mechanism included in DEVSLib, in order to facilitate the description of P-DEVS models in Modelica. The different approaches analyzed to develop the communication mechanism, as well as the description of the elements to perform the communication between models are discussed.

6.1. Message passing communication in DEVSLib

A message passing mechanism has been included in DEVSLib. In order to implement the message transmission between DEVSLib models, three different approaches have been programmed and evaluated [54]:

- *Direct transmission*, including in the connector the variables required to describe the message. This approach does not allow the simultaneous reception of several messages through the same input port.
- *Text file storage*, using a text file as intermediate storage space for the received messages. The performance of this approach is very poor due to the high amount of I/O operations needed to use the text files.
- *Dynamic memory storage*, substituting the text file with dynamically managed memory space. This approach improves the performance and offers better flexibility to manage the information of the messages, so it is the approach implemented in DEVSLib.

The dynamic memory storage approach has been implemented in C and connected with DEVSLib using the external function interface provided by Modelica. At the user level, the communication among DEVSLib models is defined by connecting the output ports of some models to the input ports of other models. The message passing mechanism is transparent to the modeler, and thus only standard Modelica *connect* sentences are used to describe the communication channels between DEVSLib models.

DEVSLib allows the user to define the type of information of each message. Messages are implemented as Modelica records. The default message type contains the following information: *Type*, represented by an integer value, and *Value*, which is represented by a real value. The message also includes a *Port* value, that represents the port the message has been received through, but this value is managed by the receiver model and not by the user. As several messages can be simultaneously sent through an output port, this type of message can be used to transmit arbitrarily complex information.

6.2. Connections between DEVSLib models

The messages are transmitted through the model connections and received by the models connected to the output ports. Each receiver model collects the arrived messages and decides which transition to execute. The simultaneous occurrence of internal and external events (i.e., a confluent event) is detected using equations and mutually exclusive boolean conditions. The event detection and management mechanism is detailed in Section 7.

The DEVSLib implementation of the input and output P-DEVS ports are the *inPort* and *outPort* connectors (see Fig. 1c). These two connectors are composed of one across variable, named *queue*, and one through variable, named *event*. An example of the communication between models in DEVSLib, using the *inPort* and *outPort* connectors, is shown in Fig. 2.

The *event* variable represents a counter of the received messages in an input port. Every time a message is sent through an output port, the *event* value of that port is increased. As *event* is a through variable, all the values of the *event* variables from the output ports connected to an input port are summed, giving the final number of messages received at that input port.

The *queue* variable represents the reference to the dynamic memory space used to temporarily store the received messages until the model executes its external transition. The messages are read, and deleted, from the memory by the external transition function. However, in order to facilitate the management of simultaneous messages, messages can be read arbitrarily – i.e., non-sequentially, using an index – without deleting them from memory.

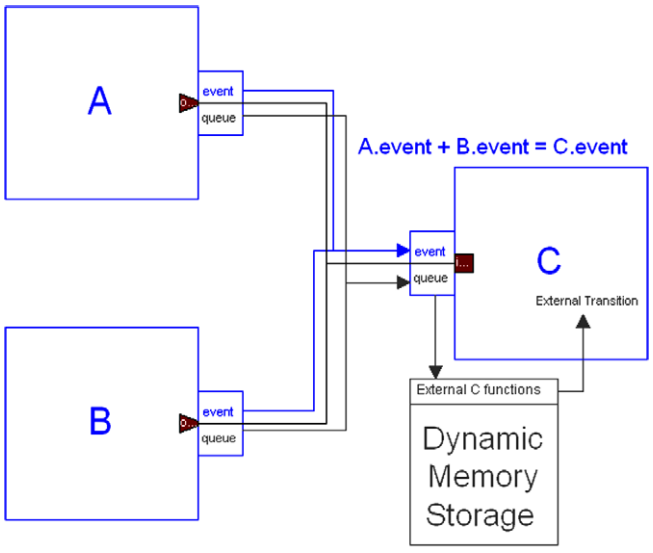

Fig. 2. Example of DEVSLib models communication scheme.

6.3. 1-to-many connections

Using the implemented message passing communication mechanism is not possible to perform 1-to-many connections between models. This limitation arises because each input port has a queue for storing incoming messages. An output port of a model connected to the input port of another model receives the reference to that queue, used to write the transmitted messages. Each output port can send messages only to one input port, because the queue variable in the connector cannot be assigned with several values (corresponding to the references of the queues that will have to receive the message).

A possible solution is the inclusion of an intermediate model to duplicate the received message and simultaneously send copies of it to several receivers. This model should have several output ports, each one connected to a receiver, that will be used to send copies of the message. Several output ports can send messages simultaneously to the same input port, because all of them share the reference to the same queue (as shown in Fig. 2). DEVSLib includes the DUP model to facilitate the 1-to-many connections. The DUP model is described in Section 8.1.

7. Atomic P-DEVS models in DEVSLib

This section describes the implementation of the behavior of a general atomic P-DEVS model in DEVSLib, and the development of new atomic P-DEVS models using the implemented behavior. The functionalities provided by Modelica to describe abstract classes, replaceable objects and functions, as well as the functionalities for event management, have been used in this implementation.

7.1. Atomic P-DEVS behavior in DEVSLib

DEVSLib includes an abstract model, named *AtomicDEVS* (see Fig. 1c), that implements the basic behavior for the atomic P-DEVS model. DEVSLib allows direct interface connections, as described in Section 4.3, by including continuous-time inputs for the transition functions to facilitate the combination of DEVSLib models with models from other Modelica libraries. The value of the continuous-time signal is read and can be used during the execution of the transition function.

The AtomicDEVS model includes the management for the internal, external and confluent events, the generation of the bag of output messages, and the sequence of actions performed during any event. Since DEVSLib has been developed under Dymola, DEVSLib uses the provided time and event management mechanisms to describe the model behavior. Only the triggering conditions for time events (usually internal events where $t_{nextInt} = t + \sigma$), the management of the messages between models, and the occurrence of simultaneous events needed to be taken into account for the development of the library.

The event detection and transition execution process performed by the AtomicDEVS model is shown in Fig. 3. The AtomicDEVS model triggers an external event when the *event* variable of any of the input ports (*iEvent[i]*) changes its value. Notice that the number of input ports is defined by the modeler, and thus a condition must be set for each port separately. Internal events are triggered when the simulation time reaches the scheduled time for the next internal transition. Confluent events are triggered as the simultaneous occurrence of both situations. Mutually exclusive boolean conditions decide which transition should be executed at each event.

During an external transition, the AtomicDEVS model updates the value of the variable that stores the elapsed time and executes the *Fext* (δ_{ext}) function, with the current state, the elapsed time and the bag of received events as parameters. After that, the state of the model is updated using the output of *Fext*. During internal transitions, the AtomicDEVS model executes the output function *Fout* (λ) using the current state, which is later updated using the output of *Fint* (δ_{int}).

During the execution of *Fout*, output messages are sent using the external function *sendEvent()*. The AtomicDEVS model checks the queues of the output ports (represented by the array *oQueue[i]*) to find if any message has been sent through them during the execution of *Fout*. If any message has been sent, the AtomicDEVS notifies the transmission of the message by increasing the value of the event variable of each port (*oEvent[i]*) by the number of messages sent through it.

In a confluent transition, the AtomicDEVS model generates an output executing the *Fout* function, using the current state. After that, it updates the variable that stores the elapsed time, executes the *Fcon* (δ_{con}) function, and updates the state of the model with its output.

A new internal transition is scheduled using *Fta* (*ta*) after each transition. The output of the Fta function can be any real number, including zero (negative numbers are considered as zero). In this way, immediate internal transitions can be scheduled. Immediate internal transitions after external or confluent events are detected by the condition described in Fig. 3. Immediate internal transitions after internal events are detected checking the value returned by *Fta* inside a while loop (because the condition described above will not detect the new internal transition, since it is the same for every internal event).

The simulation time is advanced from one internal event to the next, following the calendar of scheduled events. External events are induced by the outputs generated at internal events. Dymola manages the events in the calendar based on the conditions set for the internal events, following the procedure described in Section 3.2.

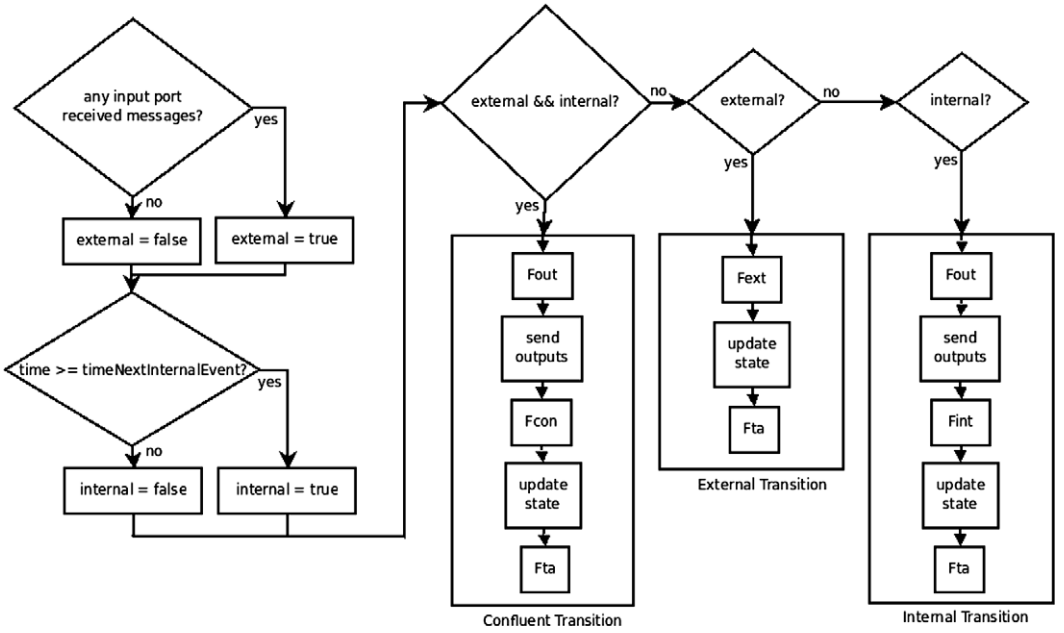

Fig. 3. Event detection and transition execution diagram of the AtomicDEVS model.

7.2. Construction of new atomic models

The description of atomic models using DEVSLib follows its formal P-DEVS specification. The user can define the state variables of the model and their initialization. The user also has to describe the actions performed by the transition functions, in order to update the state of the model after an event, as well as the time advance and output functions.

In order to construct a new atomic model, the user can duplicate the atomicDraft model (shown in Fig. 1a) and use it as a skeleton for the new model. The steps required to develop the new model are:

(1) Define the interface of the model: including the required input and output ports, as instances of the DEVSLib *inPort* and *outPort* connectors, and setting the value of the *numIn* and *numOut* parameters to the number of included input and output ports. The atomicDraft model includes by default one input and one output port. The included ports have to be linked with the *iEvent*, *iQueue*, *oEvent* and *oQueue* arrays of the AtomicDEVS model, in order to allow the correct reception and transmission of messages. This link is performed by assigning to the positions of these arrays the values of the *event* and *queue* variables of the ports. An example of these assignments is shown in Listing 1.
(2) Redefine the state: including in the *st* record the required variables to describe the state of the new model (i.e., number of customers in queue, processing units, etc.). By default, the atomicDraft includes two variables, *phase* (used to represent the current phase of the model) and *sigma* (used to schedule the next internal event).
(3) Redefine the initialization of the defined state: including in the *initst* function the initial values for the variables in *st*. The *initst* function receives the *st* record as input and returns the initialized *st* record.
(4) Redefine the transition functions: including in the *Fext*, *Fint* and *Fcon* functions the Modelica code that describes the actions performed during transitions. By default, the confluent transition function executes first the internal transition and then the external event. The internal and external transition functions return the same state by default.

```
  parameter Integer numIn = 2 "number of input ports";
  parameter Integer numOut = 1 "number of output ports";
  inPort in1 "first input port";
  inPort in2 "second input port";
  outPort out1 "first output port";
equations
  in1.event = iEvent[1];
  in1.queue = iQueue[1];
  in2.event = iEvent[2];
  in2.queue = iQueue[2];
  out1.event = oQueue[1];
  out1.queue = oEvent[1];
```

Listing 1. Assignments between interface ports and AtomicDEVS variables.

(5) Redefine the output function: including in the *Fout* function the Modelica code that generates output messages (i.e., calling the sendEvent() function). The default function does not generate any message.
(6) Redefine the time advance function: modifying the *Fta* function to return the time for the next internal transition, depending on the current state. By default it returns the value of the sigma variable of the state, that should have been previously assigned with a value during the execution of the transition function.

8. Coupled P-DEVS models in DEVSLib

Coupled DEVSLib models are described following their P-DEVS specification. A coupled model is composed of: an *interface*, that allows the connection of the coupled model with other models; its *internal components*, which are a combination of atomic or coupled models and; the *coupling connections* between the interface and the internal components, and between internal components themselves.

The interface of a DEVSLib coupled model is described using input and output ports (see Fig. 1c), which are Modelica *connectors*. The internal components of a DEVSLib coupled model are defined instantiating objects from other already available atomic or coupled DEVSLib models. Since the message passing mechanism used to communicate DEVSLib models is transparent to the user, the coupling connections between ports and components are defined using Modelica *connect* sentences between input and output ports.

The *coupledDraft* model included in DEVSLib provides a simple way to start the development of a new coupled DEVSLib model. It can be duplicated and the new copy adapted to the behavior of a new coupled model. New input and output ports can be included, by inserting new instances the DEVSLib *inPort* and *outPort* connectors. The components of the model can be included in the same fashion, instantiating the required components that have been previously developed. The coupling connections are defined by including Modelica *connect* sentences between input and output ports, either between the interface and the internal components or among the internal components themselves. Dymola offers functionalities to perform these procedures using drag and drop and is able to graphically define connections between ports.

8.1. Additional characteristics included in DEVSLib

The following additional characteristics have been included in DEVSLib to improve the construction of coupled models:

- The first characteristic concerns the simultaneous connections between the output port of one model with multiple input ports (i.e., 1-to-many connections). This problem has been described in Section 6.3. DEVSLib includes a model, named *DUP*, to reproduce 1-to-many connections. The DUP model contains one input port, used to receive messages, and two output ports, used to send copies of the received message to multiple receivers. The DUP3 model is similar to the DUP model, but has three output ports. Also, several DUP model can be serially connected if more that three copies of the message are required.
- The second characteristic concerns the generation of algebraic loops while connecting model components. An algebraic loop is generated when the output of a model is connected to the input of another model, directly or indirectly connected to the former, creating a loop between both models. As Modelica follows the synchronous data flow principle, this situation cannot be solved automatically by the simulator (it cannot find the correct causality assignment for the models in the loop) and produces an error. Similarly to the previous case, DEVSLib includes a model, named BreakLoop, aimed at avoiding this situation. The BreakLoop model defines the causality and breaks the algebraic loop by inserting a *pre()* operator in the detection of its external events. At event instants, the *pre()* operator returns the "left limit" of a variable after the last event iteration. This functionality can be used to decide which value of the variable has to be used in the calculations during the treatment of events, and to define the causality in the connections. Consider the

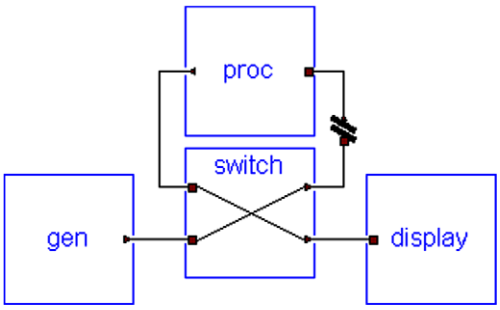

Fig. 4. Use of BreakLoop model.

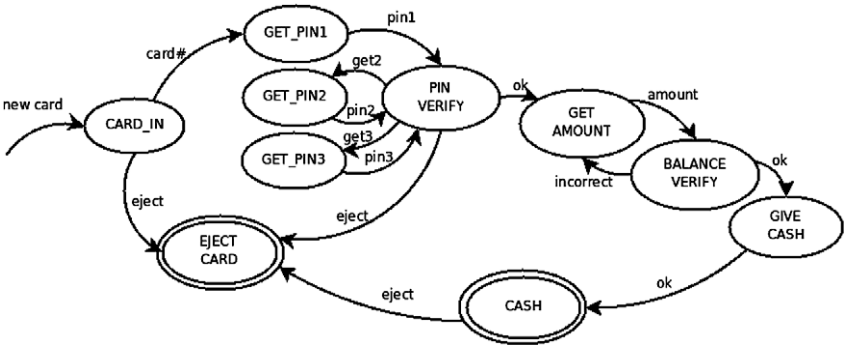

Fig. 5. State diagram of the ATM system (the system generates outputs at encircled states).

model shown in Fig. 4. The connections between the "proc" and "switch" models generate an algebraic loop, since input and output ports are internally related in both models. The BreakLoop model includes a *pre()* operator in the variable used to detect external events, and thus breaks the loop between "proc" and "switch".

– The third characteristic is the possibility to connect the output of a model to the input of the same model (i.e., self-connections). This behavior is not allowed in P-DEVS, but cannot be restricted in the Modelica environment. The modeler has to describe the model avoiding this type of connections.

9. Discrete-event system modeling with DEVSLib

The modeling of a discrete-event system using DEVSLib is discussed in this section. The model described represents an ATM system (Automatic Teller Machine) that is composed of a card reader, an operation authorization subsystem and the

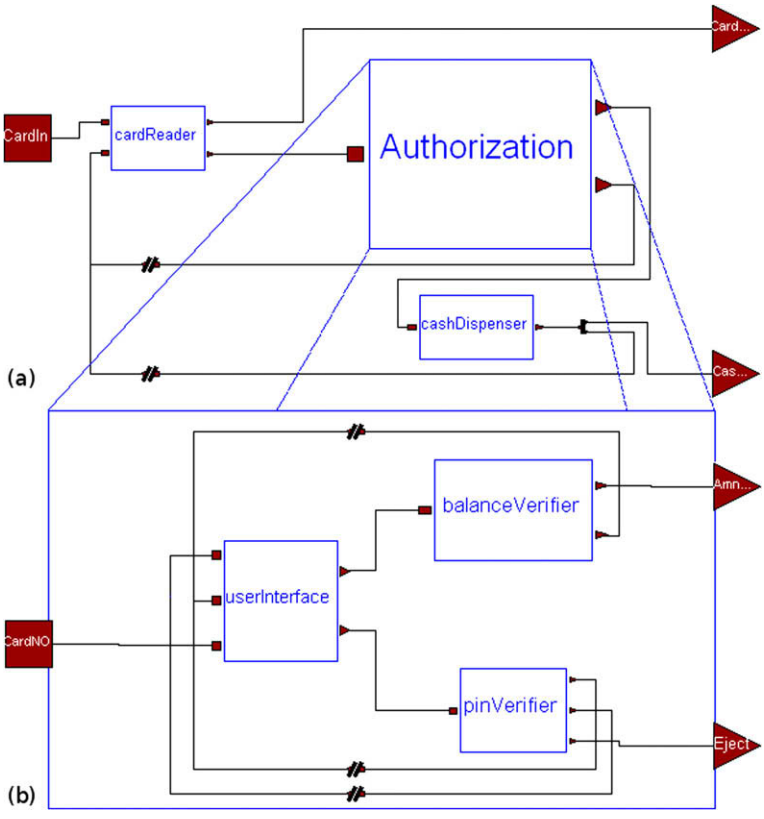

Fig. 6. ATM system modeled using DEVSLib: (a) top-level components; and (b) authorization subsystem.

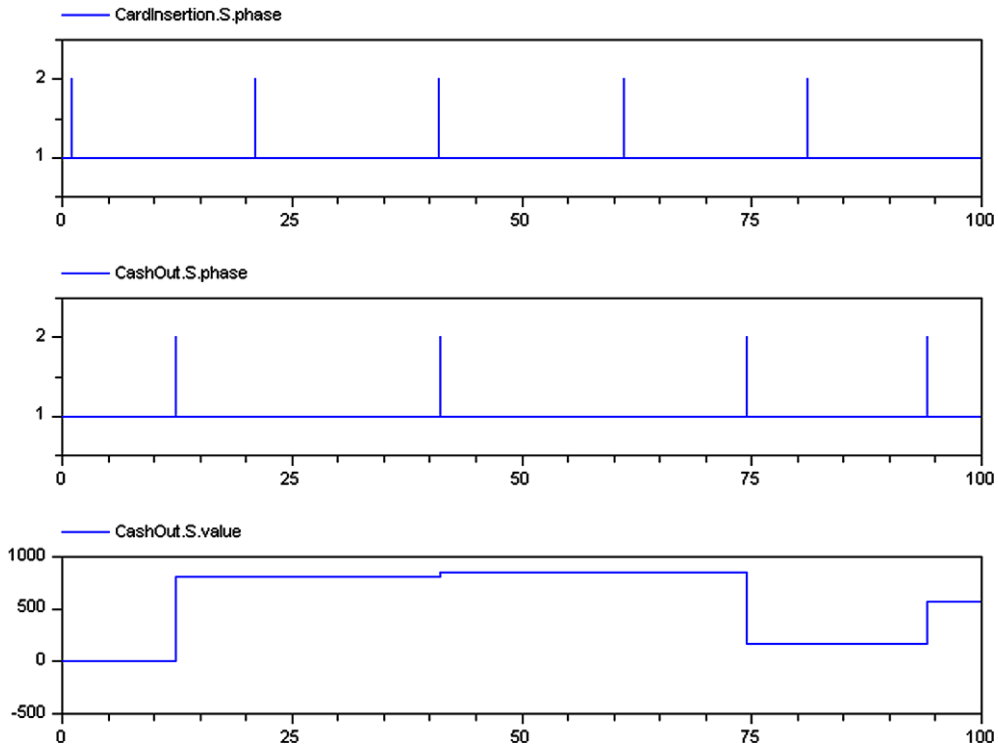

Fig. 7. Simulation results for the DEVSLib ATM model, obtained using Dymola.

cash dispenser. The behavior of the system is described in the state diagram shown in Fig. 5, and the DEVS specification of the system can be found in [20].

The user inserts a card in the ATM. The system recognizes the new insertion and asks the user to enter his PIN number. In case of an incorrect PIN number, the system asks the user again to enter the correct PIN. If the user fails thrice to enter the correct PIN, the system ejects the card. When the correct PIN is entered, the system asks the user to enter the amount of cash to withdraw. If the balance in the account of the user is insufficient, the system asks the user for a new amount. When the balance is correct, the system gives the cash to the user and ejects the card. While the system is busy, any new card insertion is ignored.

The ATM system constructed using DEVSLib is shown in Fig. 6 (notice the required *DUP* and *BreakLoop* models). The BreakLoop models are required to define the causality in the loops. The card reader and the cash dispenser are simple atomic DEVSLib models. The operation authorization mechanism is modeled using a coupled model, as shown in Fig. 6. It is composed of three atomic models: the user interface, the balance verifier and the PIN verifier. The interactions between the system and the user, in order to obtain the PIN number and the amount of cash, have been modeled statistically generating the data from random uniform distributions. The correctness of the PIN number and the balance in the user account has also been modeled using uniform random numbers.

The correspondence between the model shown in Fig. 6 and the diagram shown in Fig. 5 is as follows. The card reader performs the CARD_IN action. The GET_PIN and the GET_AMOUNT actions are performed by the user interface. The PIN_VERIFY and the BALANCE_VERIFY actions are performed by the pin verifier and balance verifier models, respectively. Finally, the GIVE_CASH action is performed by the cash dispenser. The CASH and EJECT_CARD outputs represent the output messages that arrive at the output ports in Fig. 6a.

The simulation results are shown in Fig. 7. The card insertions are shown at the top. In the center the end time of the operations, and below the amount of cash withdrawn by the user in each operation are also shown. It can be noticed that since the insertions of the card are modeled at a constant rate, some of the insertions (in this case the third one) are ignored because the system is still busy with the previous insertion.

10. Continuous-time system modeling with DEVSLib

Most numerical integration methods used in computer simulation (e.g., Euler, Runge–Kutta, DASSL, etc.) are based on time discretization. The QSS methods quantize the values of the state variables and observe the variations in their values. The quantization function used in QSS can be defined as

$$q(t) = \begin{cases} Q_m & \text{if } t = t_0 \\ Q_{k+1} & \text{if } x(t) = Q_{k+1} \wedge q(t^-) = Q_k \wedge k < r \\ Q_{k-1} & \text{if } x(t) = Q_k - \varepsilon \wedge q(t^-) = Q_k \wedge k > 0 \\ q(t^-) & \text{otherwise} \end{cases} \quad (1)$$

and

$$m = \begin{cases} 0 & \text{if } x(t_0) < Q_0 \\ r & \text{if } x(t_0) \geqslant Q_r \\ j & \text{if } Q_j \leqslant x(t_0) < Q_{j+1} \end{cases} \quad (2)$$

where Q_i are the quantization levels, $Q_i \in \{Q_0, Q_1, \ldots, Q_r\}$, usually defined using a constant quantum $(Q_{k+1} - Q_k)$. The width of the hysteresis is defined by ε.

Using this quantization function with hysteresis, a QSS system can be defined as follows. Having the following system:

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)) \\ y(t) &= g(x(t), u(t)) \end{aligned} \quad (3)$$

Its associated quantized state system is defined as

$$\begin{aligned} \dot{x}(t) &= f(q(t), u(t)) \\ y(t) &= g(q(t), u(t)) \end{aligned} \quad (4)$$

where $q(t)$ is related to $x(t)$ using the quantization function with hysteresis described above [17].

QSS systems can be described as DEVS models, combining static functions with hysteretic quantized integrators. These integrators can be described as atomic DEVS models, considering the variations in the input values as input events and generating new values as output events. The use of a quantization function with hysteresis allows to define legitimate DEVS models, avoiding problems with infinite numbers of events in a finite time interval [17,27].

10.1. QSS methods in DEVSLib

Three QSS methods have been implemented as atomic DEVSLib models: the first (QSS1), second (QSS2) and third (QSS3) order algorithms. The QSS2 and QSS3 methods need to communicate the first and second derivative together with the state value. Since the default message type transmits only one real value, several messages are simultaneously sent in QSS2 and QSS3. The *Type* variable is used to identify whether the transmitted value corresponds to the state (*Type* = 1), its first derivative (*Type* = 2) or its second derivative (*Type* = 3).

10.2. Case study

The Lotka–Volterra model of the predator–prey interaction [21,22] is used to illustrate the continuous-time system modeling with DEVSLib. The equations of the Lotka–Volterra model are the following:

$$\begin{aligned} \frac{dx}{dt} &= x\alpha - xy\beta \\ \frac{dy}{dt} &= -y\gamma + xy\delta \end{aligned}$$

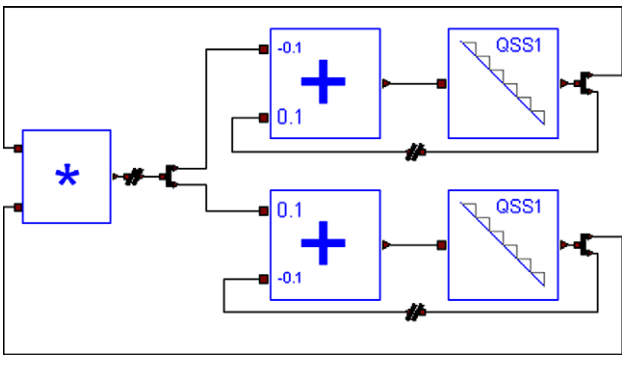

Fig. 8. Lotka–Volterra model composed using DEVSLib.

where y is the number of predators, x is the number of preys, and α, β, γ and δ are parameters that represent the interaction between both species (in this case study the value $\alpha = \beta = \gamma = \delta = 0.1$ has been used). The predator and the prey populations are inversely related: the growth of one of the species reduces the growth rate of the other, and vice versa. The result is an oscillatory behavior in the population of both species.

The model described using DEVSLib QSS algorithms is shown in Fig. 8, using the first order integrator (QSS1). The QSS1 model could be substituted with either the QSS2 or QSS3 models to apply other integrator. The multiplier and adder modules are also DEVSLib atomic models. Three DUP models are required to divide the flow of messages at the output of the integrators and the multiplier. Also, three BreakLoop models are included to break the algebraic loops between the adder, the multiplier and the integrators.

In order to compare the simulation results and performance, the Lotka–Volterra model has also been developed using the PowerDEVS software tool and the ModelicaDEVS library. The simulation results obtained by using DEVSLib, PowerDEVS and ModelicaDEVS are shown in Fig. 9. The model has been simulated using the QSS1 (Fig. 9a), QSS2 (Fig. 9b) and QSS3 (Fig. 9c) methods. The results using QSS1 in the three implementations almost overlap (see the left side of Fig. 9a). The results using QSS2 and QSS3 in the PowerDEVS and DEVSLib models are also very similar (see the left side of Fig. 9b and c). The results obtained with the ModelicaDEVS model using QSS2 and QSS3 are different from the other models. The most likely cause of

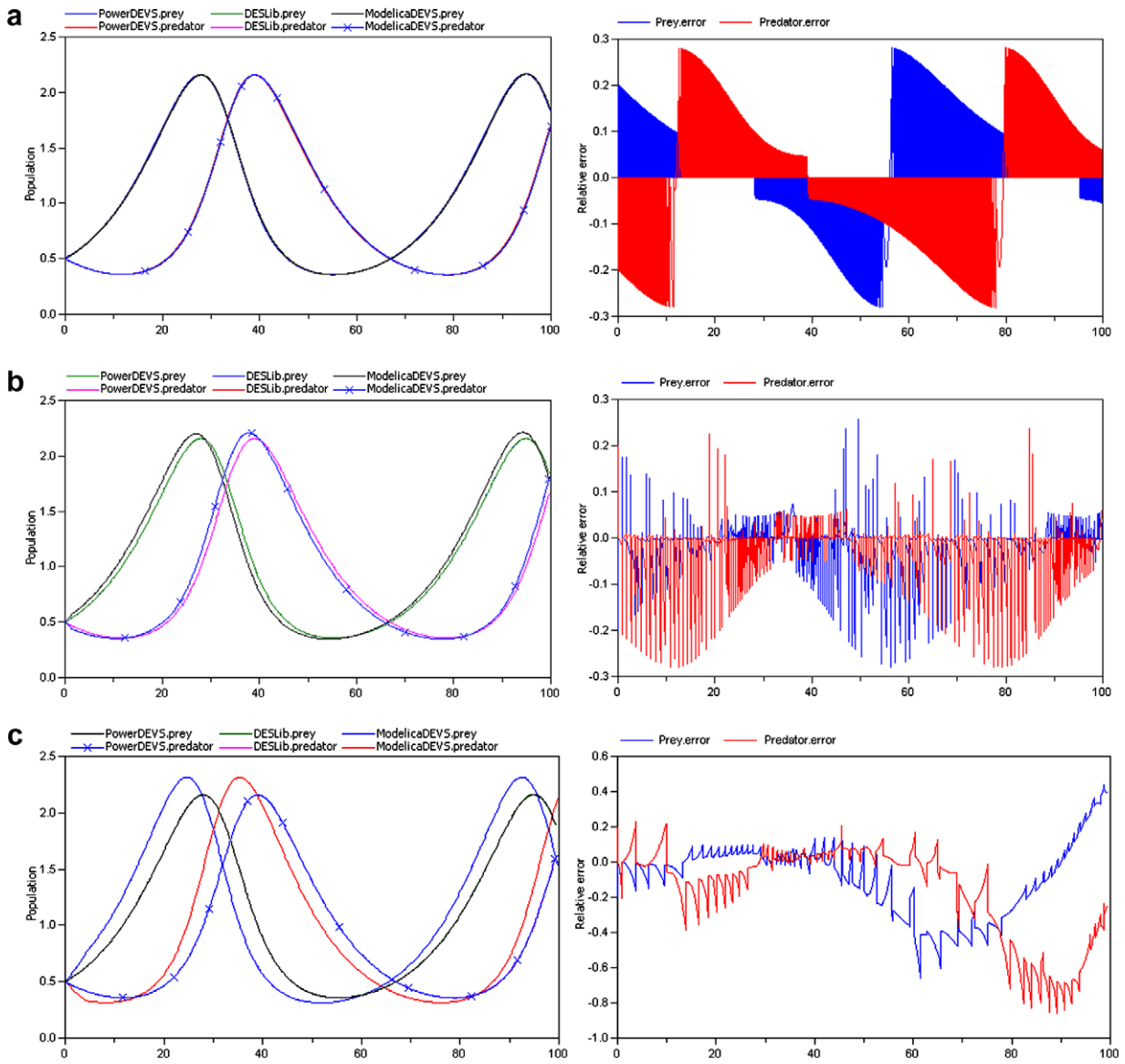

Fig. 9. Simulation of the Lotka–Volterra model developed using DEVSLib, PowerDEVS and ModelicaDEVS (relative errors between the PowerDEVS and DEVSLib models at the right). Integration method: (a) QSS1; (b) QSS2; and (c) QSS3.

Table 1
Comparison of simulation performance based on the Lotka–Volterra model.

	QSS1	QSS2	QSS3
DEVSLib			
Execution time (s)	2.19	0.078	0.031
Number of events	17,366	509	153
PowerDEVS			
Execution time (s)	1.19	0.022	0.0047
Number of events	5238	172	47
ModelicaDEVS			
Execution time (s)	1.26	0.071	0.047
Number of events	15,538	490	152

these differences is a programming error in the ModelicaDEVS integrator, which has not been detected in previous evaluations using other models.

The relative errors, in percentages, between the DEVSLib and the PowerDEVS models are shown at the right side of Fig. 9a–c. The errors show the differences between the outputs of each integrator (i.e., QSS1, QSS2 and QSS3, for predators and preys). These differences remain similar when increasing the order of the integrator. The differences concerning the ModelicaDEVS implementation have not been calculated due to the aforementioned error in the implementation.

The simulation performance of the Lotka–Volterra model, using QSS1, QSS2 and QSS3, has been compared. The obtained results are shown in Table 1. The performance indicators are the mean execution time, calculated from six simulation runs, and the number of events. The simulated time is 100 s.

The best performance is obtained using PowerDEVS, as also stated in the comparison performed in [14], because it is designed for simulating discrete-event systems following the DEVS simulator described in [27]. Dymola is designed to efficiently simulate continuous-time systems, and includes algorithms to detect and treat discrete-events. This leads to a robust hybrid system simulation approach. However, these algorithms unnecessarily degrade the performance while simulating pure discrete-event systems.

ModelicaDEVS has been specifically designed for modeling of continuous-time systems using the QSS integration methods. In contrast, DEVSLib has been designed to support the P-DEVS formalism and the QSS methods have been developed by applying the facilities provided by DEVSLib to describe general-purpose atomic P-DEVS models. As observed in the simulation results, the performance of both libraries is similar.

11. Hybrid system modeling with DEVSLib

DEVSLib provides interfaces to combine continuous-time models and P-DEVS models, which translate continuous-time signals into event trajectories (i.e., series of messages), and vice versa. These interface models allow combining the use of P-DEVS models developed with DEVSLib and hybrid models developed using other Modelica libraries. It has to be noticed that these interface models are designed for the type of message defined by-default in DEVSLib (see Section 6). Similar interfaces can be developed for messages containing other types of information. Additional information about this procedure is provided in the library documentation.

The *signal-to-message* interfaces translate continuous-time signals into event trajectories, where each event corresponds with the transmission of a message. Two different implementations of this interface are included in DEVSLib: quantization and value-crossing interfaces (see *Quantizer*, *CrossUP* and *CrossDOWN* models in Fig. 1b). The quantization interface generates an event (i.e., a message) for every change in the continuous-time signal bigger than a given quantum value. The value-crossing interface generates an event every time the continuous signal crosses a given threshold in one direction, upwards or downwards.

The *message-to-signal* interface translates the received message values (i.e., the *Value* variable of received messages) into a piecewise-constant real signal. A boolean output is also included, together with the real signal output, in order to notify the reception instant of the messages. This boolean output may be useful when the received messages have the same value and consequently the reception instants cannot be inferred from the real signal output. This interface is implemented by the *DICO* model (see Fig. 1b).

11.1. Case study

An example provided in the StateGraph Modelica library [9] will be employed to illustrate the use of the DEVSLib interfaces and to compare the performance of these two libraries (i.e., StateGraph and DEVSLib). Other examples of hybrid systems modeled using DEVSLib are described in [53,55].

The model consists of two tanks interconnected with valves, which are manipulated by a discrete-event controller. One of the valves is connected to the input flow of the first tank. The output of the first tank is connected to the input of the second tank, with a valve in between to control the flow between both tanks. The third valve is connected to the output of the sec-

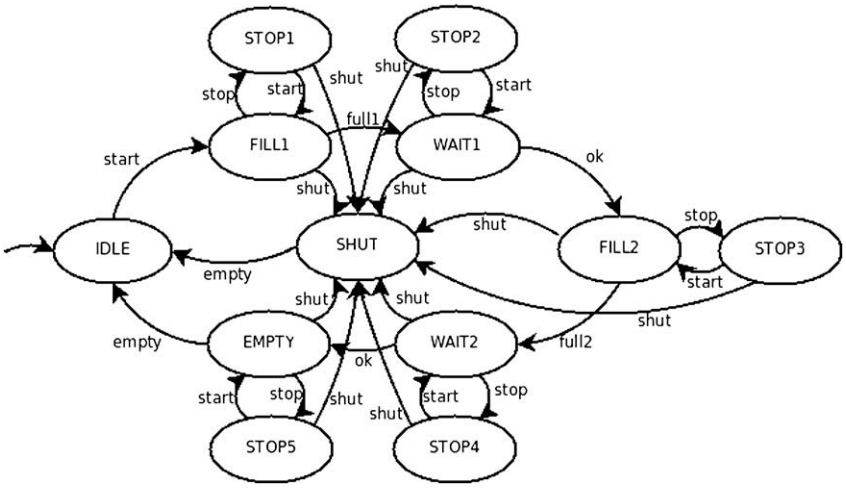

Fig. 10. State diagram of the controlled two-tank system.

ond tank. The discrete controller receives the level of each tank and controls the positions of the valves (i.e., open/close), in order to fill or empty them.

The normal operation of the system is as follows (summarized in the state diagram shown in Fig. 10):

(1) Valve 1 is opened and tank 1 is filled (the system changes from IDLE to FILL1 state).
(2) When tank 1 reaches its limit, valve 1 is closed (changing from FILL1 to WAIT1).
(3) After a waiting time, valve 2 is opened and the fluid flows from tank 1 into tank 2 (changing from WAIT1 to FILL2).
(4) When tank 1 reaches its limit, valve 2 is closed (changing from FILL2 to WAIT2).
(5) After a waiting time, valve 2 is opened and the fluid flows out of tank 2 (changing from WAIT2 to EMPTY).
(6) When tank 2 is empty, valve 3 is closed (going back to IDLE again).

Three buttons allow starting, resuming, stopping or aborting the normal operation procedure:

– *Start*, starts the process (leaving the IDLE state). When it is pressed after "stop" or "shut" the process continues (changing the state from STOP to its previous state, or restarting the normal operation procedure, respectively).

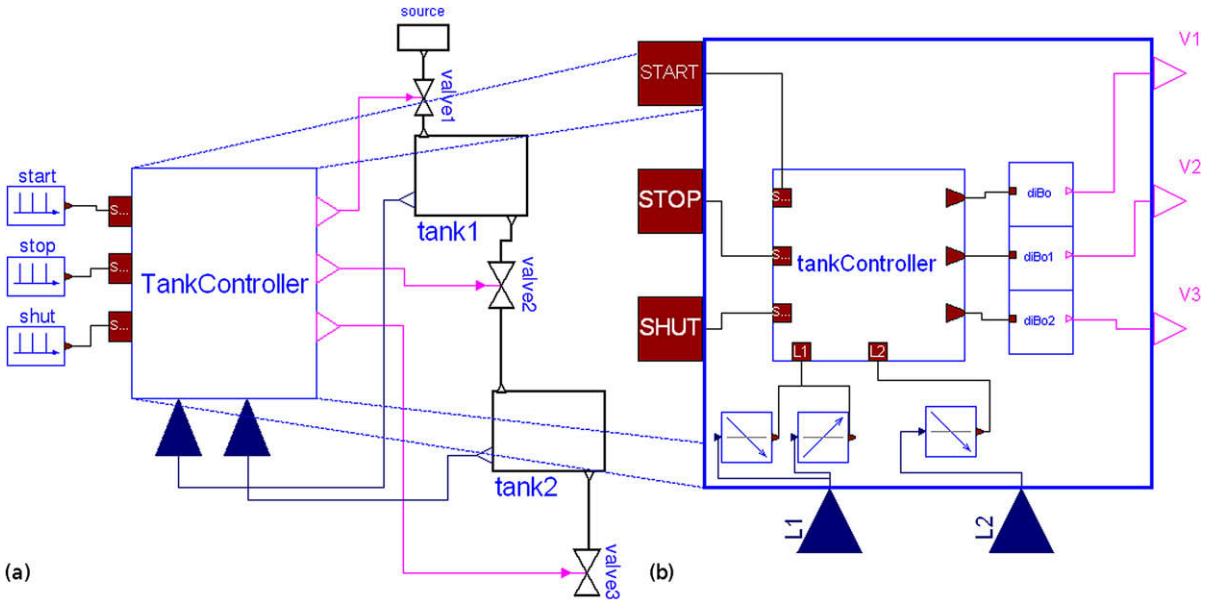

Fig. 11. Tank system modeled using DEVSLib: (a) system; and (b) controller.

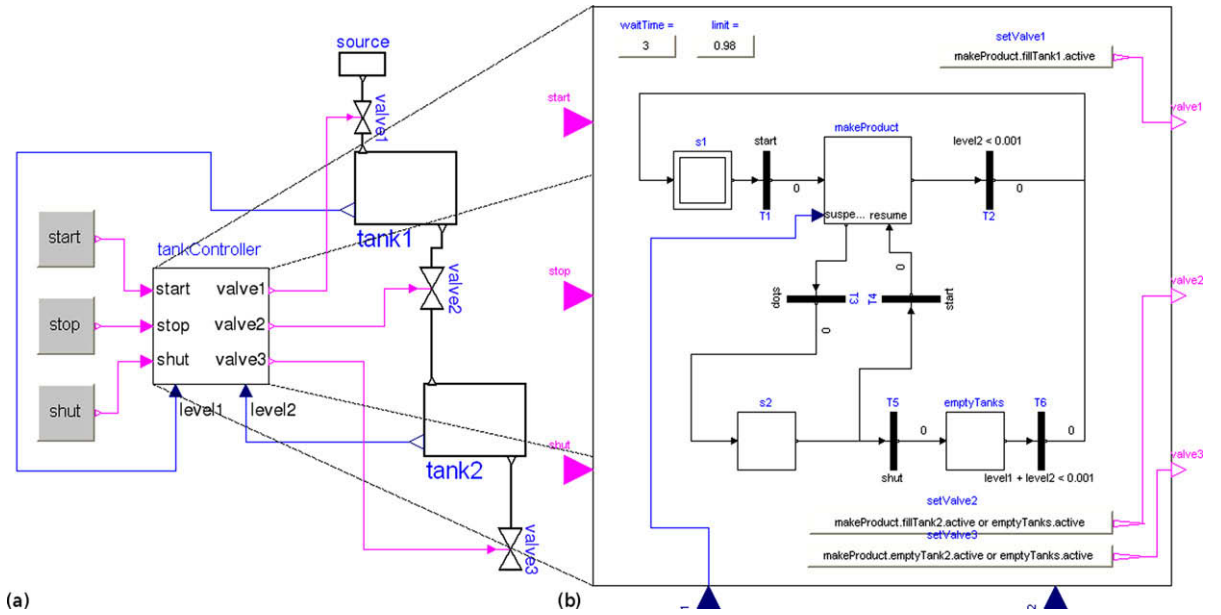

Fig. 12. Tank system modeled using StateGraphs: (a) system; and (b) controller.

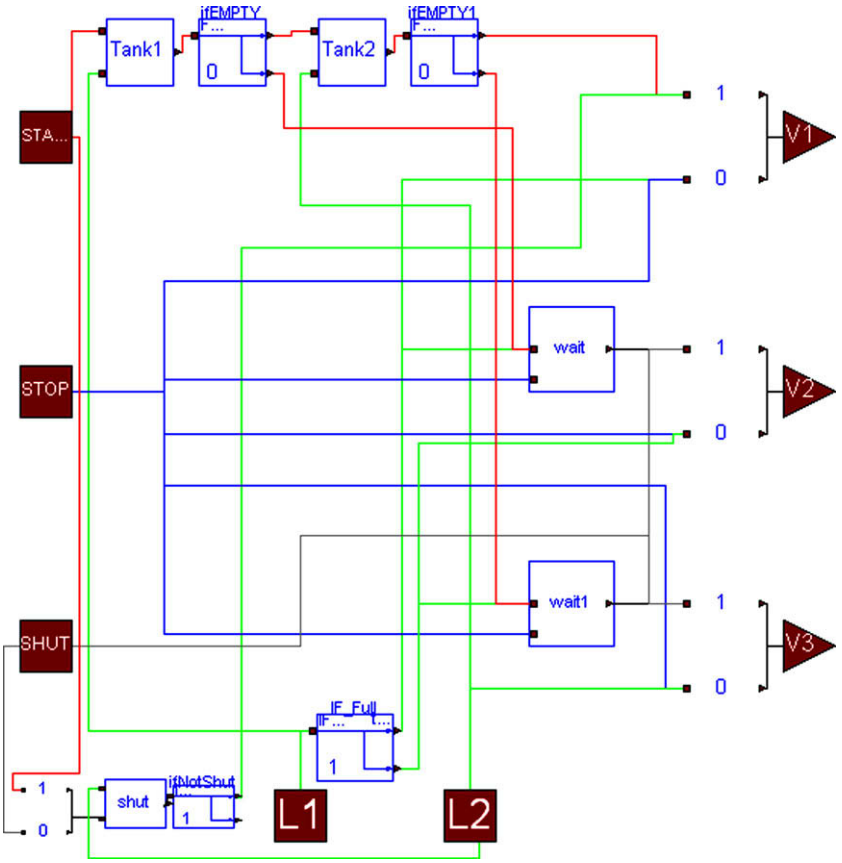

Fig. 13. Internal structure of the tank controller implemented using a coupled DEVSLib model.

– *Stop*, stops the process by closing all valves (changing to the corresponding STOP state). The controller waits for further input (“start” or “shut”).
– *Shut*, is used to shutdown the process, by emptying at once both tanks (changing to the SHUT state, and when empty changing to IDLE). After emptying the system goes to the start configuration and waits.

The diagrams of the models developed using DEVSLib and StateGraphs are shown in Fig. 11a and Fig. 12a. The continuous-time part (i.e., the tanks and valves) is the same in both models. Its components (source, valves and tanks) were developed using plain Modelica code, and can be later interconnected to describe the structure of the system.

The internal structure of the controllers is shown in Fig. 11b and Fig. 12b. The StateGraph controller implements the states and the transitions needed to achieve the desired plant operation. The controller implemented with DEVSLib includes the models to translate the continuous-time signals from the tanks, L1 and L2, into trajectories of events. The level of tank 1 is translated with two cross value models, one for detecting the full level (set to 0.98 m) and another for the empty level (set to 0.001 m). Tank 2 only needs the detection of the empty level. Also, the controller outputs are translated into boolean signals (V1, V2 and V3), that control the state of the valves. These discrete-to-boolean models behave exactly like the described discrete-to-continuous models, but generating a boolean signal instead of a real signal.

The controller itself is a P-DEVS coupled model, shown in Fig. 13 (all the required *DUP* models have been removed from the figure to improve its readability), that implements the described logic using small P-DEVS atomic operations included in the library (ifType, storage, setValue, etc.). Also, the DEVSLib controller can be implemented as a P-DEVS atomic model including the control algorithm in the transition functions. The P-DEVS specification of these models is detailed in the documentation of the model included in the library. The simulation results of both models are identical (see Fig. 14).

The simulation performance of the models composed using DEVSLib and StateGraphs has been evaluated. Two different DEVSLib implementations of the controller have been considered: first implementing the controller as an atomic P-DEVS model and second implementing it as a coupled P-DEVS model. The models have been configured to continue with the normal operation process during the whole simulation time, because the initial configuration stops the normal operation around time 24 s. Again, the performance indicators are the mean execution time, calculated from six simulation runs, and the number of events. The simulated time is 1000 s. The performance comparison is shown in Table 2.

It can be noticed that the DEVSLib model with the atomic controller and the StateGraph model have similar execution times. The simulation of the coupled DEVSLib controller consumes more time than the simulation of the atomic DEVSLib

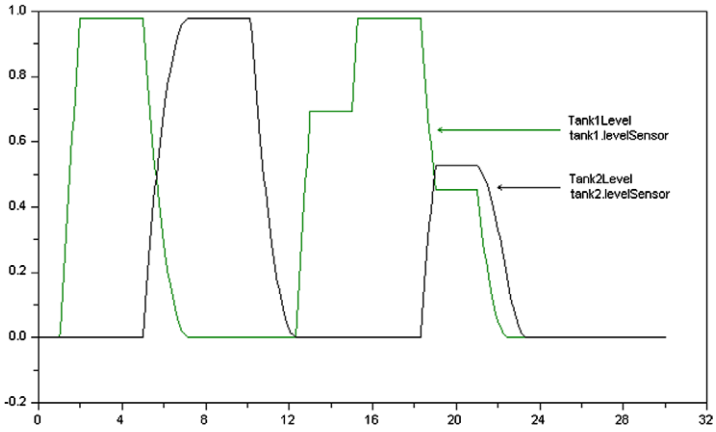

Fig. 14. Simulation results of the tank filling/emptying system (DEVSLib and StateGraph results overlap).

Table 2
Performance comparison based on the tank system.

DEVSLib (coupled)	
Execution time (s)	0.313
Number of events	170 (time) + 448 (state)
DEVSLib (atomic)	
Execution time (s)	0.078
Number of events	168 (time) + 446 (state)
StateGraphs	
Execution time (s)	0.094
Number of events	168 (time) + 447 (state)

controller. This difference in performance, even having similar number of events, is mainly due to the amount of operations performed during each event. The coupled controller activates multiple algorithms while the atomic controller has only one. However, the coupled DEVSLib controller is easier to understand than the atomic DEVSLib controller.

12. Conclusions

A free Modelica library for discrete-event system modeling, using the P-DEVS formalism, has been presented. The description of an atomic P-DEVS model using DEVSLib is very close to its formal specification – i.e., it is performed by describing each element of the tuple. This facilitates the model description and the understanding of the developed models.

The description of a coupled P-DEVS model with DEVSLib also corresponds completely with its formal specification. It is performed simply by connecting the ports of the component P-DEVS models. The communication mechanism (i.e., the message passing mechanism) between models is transparent to the user. As the P-DEVS model connection is conceptually different from the model connection in the Modelica language, it has been necessary to propose and implement the message transmission mechanism between P-DEVS models. Different alternatives have been evaluated in terms of their flexibility and performance. The implemented solution is based on storing the transmitted messages in dynamic memory.

The user is allowed to define the type of information transmitted in the messages. A default type of message is defined in DEVSLib, which allows transmitting arbitrarily complex information between P-DEVS models. For this type of message, DEVSLib provides interfaces between DEVS and continuous-time models. In addition, DEVSLib includes models implementing QSS integration methods (i.e., QSS1, QSS2 and QSS3), which can be used to simulate continuous-time models.

The modeling capabilities of DEVSLib have been illustrated by means of three application examples. Firstly, the discrete-event model of an Automatic Teller Machine (ATM) has been employed to illustrate the development of atomic and coupled DEVS models using DEVSLib.

Secondly, the Lotka–Volterra model of predator–prey interactions has been described using the QSS integration methods supported by DEVSLib. This model has also been developed using PowerDEVS and ModelicaDEVS, and the simulation performance of the three models has been compared. PowerDEVS performs better than the Modelica-based implementations, because it is specially designed to simulate DEVS models. The ModelicaDEVS implementations of QSS methods are equivalent in performance to the DEVSLib implementations. The main advantage of DEVSLib is that the QSS methods have been implemented as P-DEVS models, which facilitates their understanding and modification.

Finally, the hybrid model of a two-tank system controlled by a discrete-event controller has been employed to illustrate the use of the DEVSLib interfaces. Three different implementations of the discrete-event controller have been compared: an atomic P-DEVS model, a coupled P-DEVS model and a stategraph model. The P-DEVS models have been developed using DEVSLib and the stategraph model using the StateGraph Modelica library. The execution time of the atomic DEVS and the stategraph implementations of the controller are similar. The coupled P-DEVS controller runs four times slower. This performance difference is due to the activation and execution of multiple components in the coupled controller during each event, while the atomic controller only executes one algorithm.

The capabilities provided by DEVSLib to define P-DEVS models are similar to the ones in the simulation environments specifically designed for supporting the P-DEVS formalism. However, these environments do not facilitate the model description by combining different modeling formalisms, and the continuous-time part of the hybrid models has to be described applying DEVS-based techniques. The main advantage of DEVSLib is that it can be used together with other Modelica libraries in order to compose multi-domain and multi-formalism hybrid models. In particular, the continuous-time part of hybrid models can be described using the Modelica state-of-the-art capabilities and the complete model can be simulated using any of the state-of-the-art Modelica environments (e.g., Dymola).

Acknowledgement

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

References

[1] A.C.H. Chow, B.P. Zeigler, Parallel DEVS: a parallel, hierarchical, modular, modeling formalism, in: Proceedings of the 26th Winter Simulation Conference, San Diego, CA, USA, 1994, pp. 716–722.
[2] B.P. Zeigler, Y. Moon, D. Kim, J.G. Kim, DEVS-C++: a high performance modelling and simulation environment, in: Proceedings of the 29th Annual Hawaii International Conference on System Sciences, Maui, HI, USA, 1996, pp. 350–359.
[3] J. Nutaro, Adevs – a discrete event system simulator, Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson, 1999, <http://www.ece.arizona.edu/nutaro/index.php>.
[4] B.P. Zeigler, H.S. Sarjoughian, Introduction to DEVS modeling & simulation with JAVA: developing component-based simulation models, 2003, <http://www.acims.arizona.edu/PUBLICATIONS/>.
[5] Q. Liu, G. Wainer, Parallel environment for DEVS and Cell-DEVS models, Simulation 86 (6) (2007) 449–471.
[6] Modelica Language Specification 3.1, 2009, <http://www.modelica.org/documents>.
[7] S.E. Mattsson, M. Otter, H. Elmqvist, Modelica hybrid modeling and efficient simulation, in: Proceedings of the 38th IEEE Conference on Decision and Control, Phoenix, AZ, USA, 1999, pp. 3502–3507.
[8] Modelica free and comercial libraries, 2009, <http://www.modelica.org/libraries>.
[9] M. Otter, K.-E. Årzén, I. Dressler, StateGraph – a Modelica library for hierarchical state machines, in: Proceedings of the 4th International Modelica Conference, Hamburg, Germany, 2005, pp. 569–578.

[10] P.J. Mosterman, M. Otter, H. Elmqvist, Modelling Petri Nets as local constraint equations for hybrid systems using Modelica, in: Proceedings of the Summer Computer Simulation Conference, Reno, NV, USA, 1998, pp. 314–319.
[11] F.E. Cellier, A. Nebot, The Modelica bond graph library, in: Proceedings of the 4th International Modelica Conference, vol. 1, Hamburg, Germany, 2005, pp. 57–65.
[12] S. Robinson, R.E. Nance, R.J. Paul, M. Pidd, S.J. Taylor, Simulation model reuse: definitions, benefits and obstacles, Simulation Modelling Practice and Theory 12 (7-8) (2004) 479–494.
[13] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley-IEEE Computer Society Press, 2003.
[14] T. Beltrame, F.E. Cellier, Quantised state system simulation in Dymola/Modelica using the DEVS formalism, in: Proceedings of the 5th International Modelica Conference, Vienna, Austria, 2006, pp. 73–82.
[15] T. Beltrame, Design and Development of a Dymola/Modelica Library for Discrete Event-oriented Systems using DEVS Methodology, Master's Thesis, ETH Zürich, March 2006.
[16] E. Kofman, Discrete event simulation of hybrid systems, SIAM Journal on Scientific Computing 25 (5) (2004) 1771–1797.
[17] F.E. Cellier, E. Kofman, Continuous System Simulation, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[18] Euclides web-site, 2008, <http://www.euclides.dia.uned.es/>.
[19] V. Sanz, A. Urquia, S. Dormido, Parallel DEVS and process-oriented modeling in Modelica, in: Proceedings of the 7th International Modelica Conference, Como, Italy, 2009, pp. 96–107.
[20] H. Saadawi, Sysc-5807 methodological aspects of modeling and simulation: assignment 1, 2004, <http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm>.
[21] A.J. Lotka, Elements of Physical Biology, Williams and Wilkins, Baltimore, 1925.
[22] V. Volterra, Variations and fluctuations of the numbers of individuals in animal species living together, in: R. Chapman (Ed.), Animal Ecology, McGraw-Hill, New York, 1931, pp. 409–448.
[23] E. Kofman, M. Lapadula, E. Pagliero, PowerDEVS: a DEVS-based Environment for Hybrid System Modeling and Simulation, Tech. Rep., LSD0306, LSD, UNR (2003).
[24] I. Dressler, Code Generation from JGraphchart to Modelica, Master's Thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, March 2004.
[25] A.B. Dynasim, Dymola dynamic modeling laboratory user's manual, 2006, <http://www.dymola.com/>.
[26] A.C.H. Chow, Parallel DEVS: a parallel, hierarchical, modular modeling formalism and its distributed simulator, Transactions of the Society for Computer Simulation International 13 (2) (1996) 55–67.
[27] B.P. Zeigler, T.G. Kim, H. Praehofer, Theory of Modeling and Simulation, Academic Press, Inc., Orlando, FL, USA, 2000.
[28] A. Jeandel, F. Boudaud, E. Larivire, ALLAN Simulation release 3.1 description, M.DGIMA.GSA1887, GAZ DE FRANCE, DR, Saint Denis La plaine, France, 1997.
[29] H. Elmqvist, A Structured Model Language for Large Continuous Systems, Ph.D. thesis, Department of Automatic Control, Lunk Institute of Technology, Lund, Sweden, 1978.
[30] P. Sahlin, A. Brign, E.F. Sowell, The Neutral Model Format for Building Simulation (v. 3.02), Tech. Rep., Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, 1996.
[31] P. Fritzson, L. Viklund, D. Fritzson, J. Herber, High-level mathematical modelling and programming, IEEE Software 12 (4) (1995) 77–87.
[32] M. Andersson, Omola – An Object-oriented Language for Model Representation, Tech. Rep., TFRT 7417, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1989.
[33] A.P.J. Breuneuse, J.F. Broenink, Modeling mechatronic systems using the SIDOPS+ language, Simulation Series 29 (1) (1997) 301–306.
[34] M. Kloas, V. Friesen, M. Simons, Smile – a simulation environment for energy systems, System Analysis Modelling Simulation 18–19 (1995) 503–506.
[35] Dassault Systemes, Computer aided three dimensional interactive application, 2009, <http://www.catia.com/>.
[36] LMS International, Imagine.Lab AMESim, 2009, <http://www.lmsintl.com/imagine-amesim-intro>.
[37] Maplesoft, MapleSim, 2009, <http://www.maplesoft.com/products/maplesim/>.
[38] MathCore Engineering AB, MathModelica System Designer, 2009, <http://www.mathcore.com/products/mathmodelica/>.
[39] ITI GmbH, SimulationX, 2009, <http://www.simulationx.com/>.
[40] P. Fritzson, P. Aronsson, P. Bunus, V. Engelson, L. Saldamli, H. Johansson, A. Karstrm, The open source Modelica project, in: Proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany, 2002, pp. 297–306.
[41] S.L. Campbell, J.-P. Chancelier, R. Nikoukhah (Eds.), Modeling and Simulation in Scilab\Scicos, Springer, New York, NY, USA, 2006.
[42] Modelica, Modelica standard library, November 2008, <http://www.modelica.org/libraries/Modelica>.
[43] H. Elmqvist, F.E. Cellier, M. Otter, Object-oriented modeling of hybrid systems, in: Proceedings of the European Simulation Symposium, Delft, The Netherlands, 1993.
[44] M. Otter, H. Elmqvist, S.E. Mattsson, Hybrid modeling in Modelica based on the synchronous data flow principle, in: Proceedings of the 10th IEEE International Symposium on Computer Aided Control System Design, Kohala Coast, HI, USA, 1999, pp. 151–157.
[45] F.E. Cellier, H. Elmqvist, Automated formula manipulation supports object-oriented continuous-system modeling, IEEE Control Systems 13 (2) (1993) 28–38.
[46] H. Elmqvist, S.E. Mattsson, M. Otter, Modelica – the new object-oriented modeling language, in: Proceedings of the 12th European Simulation Multiconference, Manchester, UK, 1998, pp. 127–131.
[47] D.A. van Beek, J.E. Rooda, Languages and applications in hybrid modelling and simulation: positioning of Chi, Control Engineering Practice 8 (1) (2000) 81–91.
[48] P.L. Barton, C.C. Pantelides, Modeling of combined discrete/continuous processes, AIChE Journal 40 (6) (1994) 966–979.
[49] IEEE, Standard VHDL Analog and Mixed-signal Extensions, Tech. Rep. 1076.1, IEEE, 1997.
[50] P. Frey, D. O'Riordan, Verilog-AMS: mixed-signal simulation and cross domain connect modules, in: Proceedings of the 2000 IEEE/ACM International Workshop on Behavioral Modeling and Simulation, Washington, DC, USA, 2000, pp. 103–108.
[51] J. Kriger, Trabajo práctico 1: antiguo reloj despertador, <http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm>.
[52] W. Sun, DEVS model representing a simple automobile factory, 2001, <http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm>.
[53] M. Brière, L. Carrel, T. Michalke, F. Mieyeville, I. O'Connor, F. Gaffiot, Design and behavioral modeling tools for optical network-on-chip, in: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe, IEEE Computer Society, Washington, DC, USA, 2004, p. 10738.
[54] V. Sanz, A. Urquia, S. Dormido, Introducing messages in Modelica for facilitating discrete-event system modeling, in: Proceedings of the 2nd International Workshop on Equation-based Object-Oriented Languages and Tools, Paphos, Cyprus, 2008, pp. 83–94.
[55] V. Sanz, F.E. Cellier, A. Urquia, S. Dormido, Modeling of the ARGESIM "crane and embedded controller" system using the DEVSLib Modelica library, in: Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems, Zaragoza, Spain, 2009.