Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling

Victorino Sanz, Alfonso Urquia, Sebastian Dormido, ETSII Informática, UNED, Spain
{vsanz,aurquia,sdormido}@dia.uned.es

The work performed by the authors to provide to Modelica more discrete-event system modeling functionalities is presented. These functionalities include the replication of the modeling capacities found in the Arena environment, the SIMAN language and the DEVS formalism. The implementation of these new functionalities is included in three free Modelica libraries called ARENALib, SIMANLib and DEVSLib. These libraries also include capacities for random number and variates generation, and dynamic memory management. They are freely available for download at `http://www.euclides.dia.uned.es/`. As observed in the work performed, discrete-event system modeling with Modelica using the process-oriented approach is difficult and complex. The convenience to include a new concept in the Modelica language has been observed and is discussed in this contribution. This new concept corresponds to the model communication mechanism using messages. Messages help to describe the communication between components in a discrete-event system. They do not substitute the current discrete-event modeling capabilities of Modelica, but extend them. The proposed messages mechanism in Modelica is discussed in the manuscript. An implementation of the messages mechanism is also proposed.

SNE 18/2, August 2008

Introduction

Several Modelica libraries have been developed by the authors in order to provide to Modelica more discrete-event system modeling capabilities. The work performed is specially based in modeling systems using the process oriented approach, reproducing the modeling functionalities of the Arena simulation environment [10] in a Modelica library called ARENALib. The functionalities of the SIMAN modeling language [18], used to describe components in Arena, have also been reproduced in a Modelica library called SIMANLib. One objective of the development of this library is to take advantage of the Modelica object-oriented capabilities to modularize as much as possible the development of discrete-event system models. Also, the use of a formal specification to describe SIMANLib components helped to understand, develop and maintain them. SIMANLib blocks can be described using DEVS specification formalism [21]. Event communication in DEVS and block communication in SIMANLib match perfectly. An implementation of the Parallel DEVS formalism [23] has been developed in a Modelica library called DEVSLib, and used to describe the components in SIMANLib. All the performed work with Modelica has been developed using the Dymola modeling environment [1]. The problems encountered during the development of the ARENALib, SIMANLib and DEVSLib Modelica libraries, and the solutions applied to those problems are discussed.

The Modelica language includes several functionalities for discrete-event management, such as `if` expressions to define changes in the structure of the model, or `when` expressions to define event conditions and the actions associated with the defined events [16].

Other authors have contributed to the discrete-event system modeling with Modelica. Depending on the formalism used to define the discrete-event system, contributions can be found using finite state machines [7, 14, 17], Petri nets [15] or the DEVS formalism [2, 3, 4, 8]. On the other hand, other authors have developed tools to simulate discrete event systems in conjunction with Modelica. For example, translating models developed using a subset of the Modelica language to the DEVS formalism. The translated models are then simulated using the CD++ DEVS simulator [5]. Also, other authors describe the discrete-event system with an external tool that translates a block diagram to Modelica code [19].

All these contributions use the event-scheduling approach for describing the discrete-event systems [12]. Events are scheduled to occur in a future time instant. The simulation evolves executing the actions associated with the occurrence of the events.

Due to the difficulties and problems encountered during the development of the mentioned Modelica libraries, the convenience of introducing a new concept in Modelica has been identified. This new con-

cept will facilitate the development of discrete-event systems, extending the current Modelica capacities. This new concept is the model communication using the messages mechanism. The main characteristics and functionalities of this mechanism are also discussed in this manuscript.

1 Process-oriented modeling in Modelica

A discrete-event system modeled using the process-oriented approach is described from the point of view of entities [10]. These entities flow through the components of the system, and some processes are applied to them using the available resources of the system. Some of the information associated with the entities are the serial number, the type, the statistical indicators, the attributes, the creation time, and the processing time among others. An example of this kind of system can be a beverage manufacturing system. The entities of this system are the bottles. A tank fills bottles with the beverage. Once filled, the bottles are labeled and quality controlled before they are accepted for distribution (first and second class bottles). Bottles without the required quality are cleaned and re-labeled. The components of this kind of systems are usually stochastic. For example, the labeling and cleaning processes are modeled using the Triangular probability distribution. The quality controls are represented by two-way decisions whose percentage is based on the values of uniform random variates.

The process-oriented approach is supported by the Arena simulation environment to model discrete-event systems. Arena has *data modules*, that represent the entities, the resources, and some other static elements of the system, and *flowchart modules*, that represent the processes performed on the entities across the system. The implementation of the beverage manufacturing system using Arena is shown in Figure 1a. It is modeled as a hybrid system, because the tank is represented by a continuous time model.

Arena allows some simple hybrid modeling by describing level variables that change continuously over time, and rate variables, that represent how fast the level variable changes its value. Each pair of level/rate variables represents a differential equation that is simulated using Euler, RKF or any user-implemented integration method.

1.1 ARENAlib

ARENALib reproduces the Arena data and flowchart modules that have to be combined and connected to

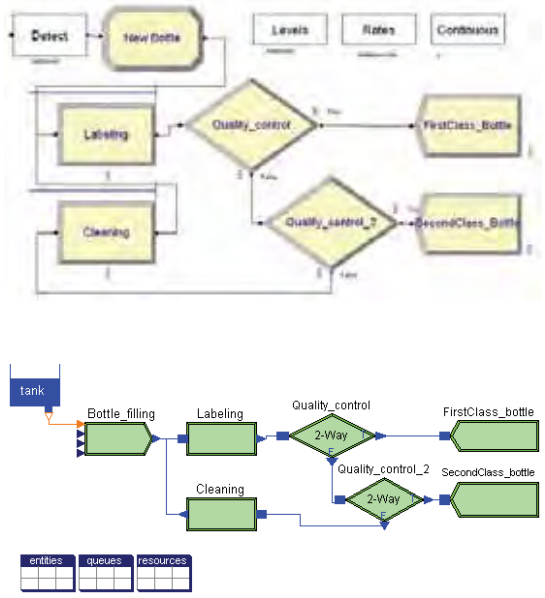

Figure 1. Beverage manufacturing system. An example of hybrid discrete-event system developed using: a) Arena; and b) ARENALib.

model the system. This library is freely available for download at [6]. At the moment, the Create, Process, Dispose and Decide flowchart modules and the Entity, Queue, Resource and Variable data modules, of the Arena Basic Process panel, have been implemented.

The library also allows hybrid system modeling, combining the current Modelica continuous-time system modeling functionalitieswith the components of ARENALib.A detailed description of the library can be found in [20]. The model of the beverage manufacturing system composed using ARENALib is shown in Figure 1b. In this figure, the *Bottle_filling* module corresponds to a Create module, *Labeling* and *Cleaning* correspond to Process modules, *Quality_control* and *Quality_control_2* are Decide modules and the *FirstClass_bottle* and *SecondClass_bottle* are Dispose modules. *Entities*, *queues* and *resources* contain the data modules required for this system.

The main tasks accomplished during the development of the ARENALib library were: a) the model communication mechanism; b) the entity management; c) the management of the statistical information and; d) the generation of stochastic data. These tasks and the solutions proposed and implemented to the problems encountered during the development of the ARENALib library are discussed below.

1.2 Model communication mechanism

Entities are generated in the system during the simulation, flow across the components of the system and, if necessary, are disposed. Generally, the number of entities in the system changes during the simulation run, depending on the behavior of the system.

Usually an entity arrives to a module, is processed and sent to the following module. Entity communication is an important part of the simulation process.

Model interaction in Modelica can be performed using connectors. A connector is an special class and contains some variables that are linked with the ones in another connector using a connect equation. The connect equation relates variables either equaling them, or summing them and equaling the sum to zero.

Several approaches have been studied, implemented and evaluated during the development of ARENALib in order to perform the entity transmission between modules. The approach used to perform the entity transmission is completely transparent for the end user. At the user level, the communication is just defined by connecting the output ports of some modules to the input ports of other modules. The mentioned approaches are discussed next.

Direct transmission

It consists of specifying all the variables that define a type of entity inside the connector. The values assigned to the variables of one connector represent an entity. These values are assigned, because of the connect equation, to the connector of the next model. In this way, an entity is directly transmitted from one model to another. Different types of entities require different connectors, one for each type. This is the simplest way for communicating models, but presents a problem: the simultaneous reception of several entities at one model. There are three possible situations for this problem:

- One-to-one connection: one model sends several entities to another model at the same time.
- Many-to-one connection: several models simultaneously send one entity to another model.
- A combination of the previous cases: several models simultaneously send one, or more, entities to another model.

The two following solutions have been applied to this problem:

1. Synchronizing the entity transmission between models using semaphores. The synchronization allows the sender and receiver to manage the flow of entities between both models, using a send/ACK mechanism like in the TCP/IP communication. Thus, the sender model will send an entity to the receiver and wait for an ACK. On the other hand, the receiver model will receive entities when it is ready to process them, and only send the ACK back if still ready to continue processing more entities. A model of the semaphore synchronizationmechanism, based on a previous work by Lundvall and Fritzson [9], has been implemented and is freely available for download at [6]. A disadvantage of this solution is the performance degradation due to the event iteration that takes place during the synchronization phase of the entity transmission.
2. Including in the connector a `flow` variable that represents the number of entities sent from a model. So, the model receiving the entities will know the number of entities received, even with many senders. However, the information that describes several entities can not be transmitted simultaneously using the direct transmission approach. The variables of the connector that describe the entity can not be assigned with different values, that represent the different transmitted entities, at the same time. Anyway, the text file storage and dynamic memory storage approaches, discussed below, allow to solve this problem using the flow variable.

Text file storage

The idea is to define an intermediate storage for the transmitted entities. This storage behaves as a communication buffer between two or more modules.

The storage is implemented in a text file that stores in each line of text the information related to each transmitted entity. The connector contains a reference to the text file, its file-name, and the flow variable indicating the number of entities received. This reference is shared between the models connected to that connector, allowing them to access the file. Each module is able to receive entities, creates an storage text file and sets the reference to that file in the connector. Functions to read/write entities from/to the file have been developed. A model writes one or several entities to the file using the write function. Another function is used by the receiver to check the number

of entities in the file. When there is any entity to be read, the receiver reads the entities and processes them. Thus, this approach allows the simultaneous reception of several entities.

A disadvantage associated with this approach is the poor performance due to the high usage of I/O operations to access the files. Also, the structure of the information stored in the files is not very flexible if any additional information has to be included. If new types of entities need to be used, or the attributes of an entity have to be changed, the file management functions (i.e. read and write) have to be re-implemented to correctly parse the text file to support these new changes.

Dynamic memory storage

In order to improve the performance of the text file approach, the intermediate storage was moved from the file system to the main memory. Using the Modelica external functions interface, a library in C was created to manage the intermediate storage using dynamic memory allocation. An entity is represented in Modelica using a `record` class, and in C using its equivalent `struct` data structure. Entities are stored using linked-lists structures during their transmission from one model to another. This library is freely distributed together with the ARENALib Modelica library.

Instead of a reference to the file, the connector contains a reference to the memory space that stores the entities, together with the flow variable that indicates the number of entities received. That reference is the memory address pointing to the beginning of the linked-list. It is stored in an integer variable in the connector. Similarly to the text file approach, each model able to receive entities initializes the linked-list and sets the reference to it in the connector. Entities can be transferred to the queue using the write function, and can be extracted using the read function. Another function is used to check the availability of received entities, in order to process them.

This approach also allows the simultaneous reception of several entities. The performance is highly increased compared to the text file approach. And, the structure of the information only depends on the data structures managed by the functions. To modify any attribute or entity type, it is only necessary to change a data structure and not all the functions used to manage that structure.

1.3 Entity management

Regarding the entity management, it has to be mentioned that an additional problem appears when implementing processes that delay the entity. Arena process module can include a delay time that represents the time spent processing the entity. This delay time is usually randomly selected from a probability distribution. It has to be noticed that since the delay time is usually random, the order of the arrived entities need not correspond to the order of the entities leaving the process. These processes have to include a temporal storage for the entities that are being delayed. This problem can be solved using the text file storage or the dynamic memory storage as an additional storage for delayed entities. Due to performance reasons, the dynamic memory approach was used to manage entity storage during delays in ARENALib and SIMANLib.

Together with the initialization of the linked-queue for entity communication, a process module initializes a temporary storage, represented by a linked-list in memory, for delayed entities. The reference to that list is also stored in an integer variable. Every time the process module has to delay an entity, it stores the entity in the list using a write function. Entities are inserted in the list in increasing order, according to the time they must leave the process. The insertion of an entity in the list returns the leaving-time for the first entity in the queue. When the simulation time reaches the next leaving-time, the entity or entities leaving the process are extracted from the list and sent to the next module.

1.4 Stochastic data generation

Discrete-event models usually contain some kind of stochastic information. Random processing times, delays or inter-arrival times help to construct a more realistic model of a given system.

The Modelica language specification does not include any functionality for random number generation. Dymola, the modeling environment used to develop and test the mentioned Modelica libraries, includes two functions for generating random uniform and random normal variates [1]. The generation of random variates following other probability distributions is not covered by these random number generation functions. Also, the application of variance reduction techniques is not supported by these functions.

A random number generator (RNG) was developed by the authors. The RNG algorithm selected for its

implementation in Modelica is the same that is used in the Arena environment. This allows the validation of the ARENALib models using the Arena environment, because both use the same source of random numbers. This RNG algorithm was proposed by Pierre L'Ecuyer and is called Combined Multiple Recursive Generator. A detailed description of the RNG is given in [13].

Additionally to the implementation of the RNG, some functions for generating random variates were also developed by the authors of this manuscript. The new RNG and the random variates generation functions are packaged in a Modelica library called RandomLib, which is freely available for download at [6].

1.5 Statistical information management

Simulation results are usually reported using statistical indicators, due to the stochastic nature of discrete-event systems. Some of these statistical indicators have to be calculated during the simulation and some others at the end. The amount of data that has to be stored to calculate some of these indicators changes depending on the length of the simulation.

Modelica does not allow the declaration of variables with an undefined length or size, which are required to store the statistical data. A mechanism to declare variables of undefined length in Modelica needs to be defined, giving the possibility to increase or decrease the size of the variable during the simulation run.

This problem is very similar to the previously mentioned one about intermediate entity storage for transmission or delay management. So, the mentioned dynamic memory storage has been used in ARENALib to record the information regarding the statistical indicators of the simulation. The indicators calculated in each ARENALib module are shown in Tab. 1. Statistical indicators calculated include the number of entities arrived, the number of entities departed, processing times, the number of entities in queue, and the number of entities in the ystem, among others. The information calculated for each indicator is the mean, the maximum value, the minimum value, the final value and the number of observations. These values are updated during the simulation. On the other hand, all the intermediate values have to be recorded and used to calculate the confidence interval at the end of the simulation. A variable in Modelica stores a reference to the memory space that contains the stored data for each indicator. That space is managed using external functions written in C.

Module	Indicator	Values
Create	System.NumberIn	Obs
Process	NumberIn	Obs
	NumberOut	Obs
	VATime Per Entity	Avg, Min, Max, Final, Obs
	NVATime Per Entity	Avg, Min, Max, Final, Obs
	TotalTime Per Entity	Avg, Min, Max, Final, Obs
	Queue.NQ	Avg, Min, Max, Final
	Queue.WaitTime	Avg, Min, Max, Final, Obs
Dispose	System.NumberOut	Obs
EntityType	NumberIn	Obs
	NumberOut	Obs
	VATime	Avg, Min, Max, Final, Obs
	NVATime	Avg, Min, Max, Final, Obs
	TranTime	Avg, Min, Max, Final, Obs
	WaitTime	Avg, Min, Max, Final, Obs
	OtherTime	Avg, Min, Max, Final, Obs
	Work In Progress	Avg, Min, Max, Final

Table 1. Statistical indicators and values calculated in the ARENALib modules

1.6 SIMANLib

The first approach for the development of ARENALib was to write all its components, except the mentioned external functions and data types which are written in C, in plain Modelica code. This generated large and complex models that were difficult to understand, maintain and extend.

The idea then was to divide the actions performed by each module into simpler actions that combined will offer the same functionality than the original module.

The same structure can be observed in the Arena environment, where the modules are based and constructed using a lower level simulation language called SIMAN [18].

SIMANLib contains low-level components for discrete event system modeling and simulation. These are low-level components compared to the modules in ARENALib, which represent the high-level modules for system modeling. Flowchart modules of both libraries are shown in Figure 2. ARENALib modules can be described using a combination of SIMANLib components. For example, the process module of ARENALib is composed by the Queue, Seize, Delay and Release blocks of SIMANLib, as shown in Fig. 3.

Components in SIMANLib are divided, as well as in the SIMAN language, in two groups: blocks and elements. The blocks represent the dynamic part of the system, and are used to describe its structure and define the flow of entities from their creation to their disposal. The elements represent the static part of the system, and are used to model different components such as entities, resources, queues, etc.

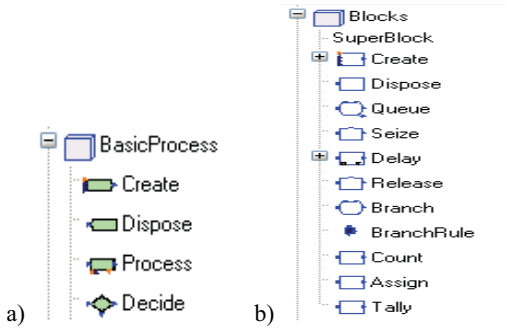

Figure 2. Flowchart modules: (a) ARENALib; and (b) SIMANLib

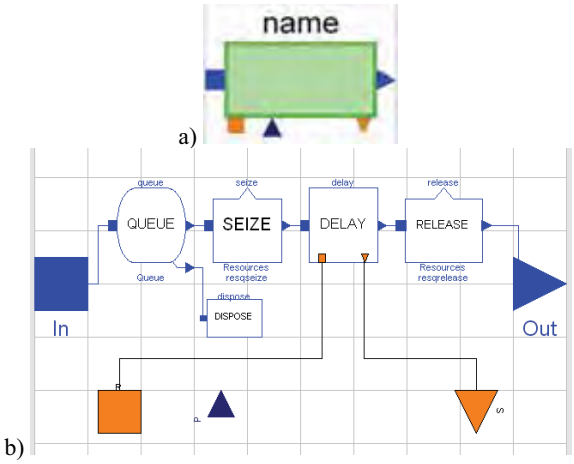

Figure 3. ARENALib process module: a) icon; b) internal structure composed using SIMANLib components.

An example of a model developed using SIMANLib is shown in Figure 4. This system is very similar to the beverage manufacturing system mentioned above. The entities are pieces to be machined. The pieces arrive to the system and are processed by a machine, one at a time. After processed, the pieces are inspected by a supervisor and classified as Good, Reject and Repair. Repaired pieces are sent back for reprocessing.

2 Parallel DEVS in Modelica

The main objective of the implementation of the DEVSLib library has been to closely follow the definition of the Parallel DEVS formalism and implement all its features without restrictions. The functionalities of DEVSLib are similar to the ones offered by other DEVS environments such as DEVSJAVA [24] or CD++ [22]. These similarities include the new atomic and coupled models construction based on predefined classes, the redefinition of the internal, external, output and time advance functions in each atomic model as required by the user and the management of model

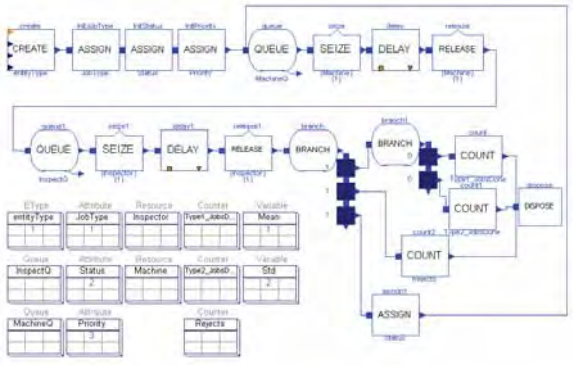

Figure 4. Manufacturing system model composed using SIMANLib components

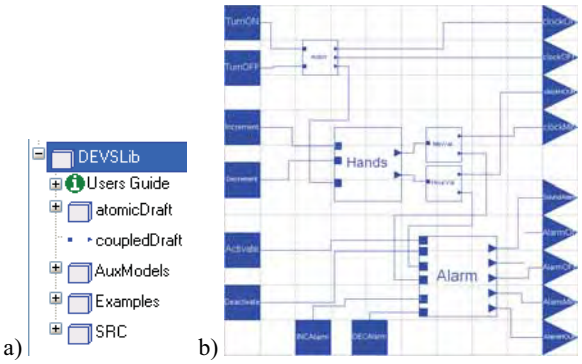

Figure 5. The DEVSLib Modelica library: a) architecture; b) case of use (model of a pendulum clock).

input and output ports as needed. However, due to the capacities of the Modelica language, DEVSLib still presents some restrictions that will be discussed below.

2.1 DEVSLib architecture

The architecture of the library is rather simple. It is shown in Figure 5a. It contains two main models, atomicDraft and coupledDraft, that represent the basic structures for building any new atomic or coupled DEVS models. Together with the main models there are several auxiliary models and functions for managing event transmission. Additionally, some examples of atomic and coupled systems have been included. One of the included examples is the hybrid model of a pendulum clock [11], which is shown in Figure 5b. In this system a continuous-time model of a pendulum generates tics, acting as the motor of the clock. The rest of the clock receives the tics, calculates the current time (in hours and minutes) and manages the alarm of the clock.

2.2 Model development with DEVSLib

When building a new atomic model, the user has to specify the actions to be performed by the external

```
model processor
  extends AtomicDEVS(redeclare record State = st);
    redeclare function Fcon = con;
    redeclare function Fint = int;
    redeclare function Fext = ext;
    redeclare function Fta = ta;
    redeclare function initState =
      initst(dt=processTime);
    parameter Real processTime = 1;
    Interfaces.outPortManager outPortManager1(
       redeclare record State = st,
       redeclare function Fout = out, n=1);
    Interfaces.outPort outPort1;      // output port
    Interfaces.inPort inPort1;        // input port
    equation
       iEvent[1] = inPort1.event;
       iQueue[1] = inPort1.queue;
       connect(outPortManager1.port, outPort1);
end processor;
```

Listing 1. Modelica code of a processor system modeled using DEVSLib

transition, internal transition, output and time advance functions. This can be performed by re-declaring the functions `Fext`, `Fint`, `Fout` and `Fta`, initially declared in the atomicDraft model. The user can specify any desired behavior for these functions, while maintaining the defined function declaration. Any new atomic model has to extend the AtomicDEVS model and to re-declare the mentioned functions. The Modelica code of a processor system [23] developed using DEVSLib is shown in Listings 1, 2 and 3.

The desired number of input and output ports can also be included in the new model and managed with the mentioned functions. The user can drag and drop new input and output ports into the model. The prototypes of the external transition and the output function allow the user to check the port where an incoming event has been received, or to specify the output port to send the event. All these ports could be connected later to other models.

A coupled DEVS model, like the one shown in Figure 5b, can be easily build using previously defined atomic or coupled models, and connecting them as required. The input and output ports have to be included and connected to any of the model components

2.3 DEVSLib modeling restrictions

One restriction in DEVSLib is the impossibility to perform one-to-many connections. These kinds of connections are not considered in ARENALib or SIMANLib because neither Arena nor SIMAN permits them. However, the Parallel DEVS formalism allows this kind of connection so they have been taken into account.

```
function con "Confluent Transtition Function"
   input st s, Real e, Integer q, Integer port;
   output st sout, soutput;
algorithm
   soutput := s;
   sout := ext(int(s),e,q,port);
end con;

function int "Internal Transition Function"
   input st s;
   output st sout;
algorithm
   sout := s;
   sout.phase := 1; sout.job := 0;
   sout.delta := Modelica.Constants.inf;
end int;

function ext "External Transition Function"
   input st s, Real e, Integer q, Integer port;
   output st sout;
protected
   Integer numreceived;
   stdEvent x;
algorithm
   sout := s;
   numreceived := numEvents(q);
   if s.phase == 1 then
      for i in 1 : numreceived loop
         x := getEvent(q);
         if i == 1 then
            sout.job := x.Value;
            Modelica.Utilities.Streams
              .print("* Event to process");
         else
            Modelica.Utilities.Streams
              .print("* Event balked");
         end if;
         sout.received := sout.received + 1;
      end for;
      sout.phase := 2;       // active
      sout.delta := s.dt;    // processing_time
   else
      sout.delta := s.delta -e;
   end if;
end ext;

function out "Output Function"
   input st s, Integer port, Integer queue;
   output Boolean send;
protected
   stdEvent y;
algorithm
   if s.phase == 2 then
      send := true;
      y.Type := 1;
      y.Value := s.job;
      sendEvent(queue,y);
   else
      send := false;
   end if;
end out;

function ta "Time Advance Function"
   input st s;
   output Real delta;
algorithm
   delta := s.delta;
end ta;
```

Listing 2. Modelica code of the functions redeclared in the processor system.

```
record st "State of the model"
  Integer phase;       // 1 = passive, 2 = active
  Real delta;          // internal transitions interval
  Real job;            // current processing job
  Real dt;             // default processing time
  Integer received;    // num of jobs received
end st;

function initst "State Initialization Function"
  input Real dt;
  output st out;
algorithm
  out.phase := 1;     // passive
  out.delta := Modelica.Constants.inf;
  out.job := 0;
  out.dt := dt;
  out.received := 0;
end initst;
```

Listing 3. Modelica code of the state and state initialization of the processor system.

This restriction appears because the way the port and the event communication mechanism is managed, using dynamic memory storage. As mentioned before, each receiver initializes its linked-queue to receive entities. A one-to-many connection cannot be performed because the sender can not store in just one integer variable the references to all the linked-queues created by the receivers. A solution has been implemented in the DEVSLib library. This solution consists in an intermediate model that can be used to duplicate the events and send them to the receivers. Examples of this intermediate model are the *MinValue* and the *HourValue* models shown in Figure 5b.

By default, the information transmitted between models in DEVSLib, at event instants, is composed by two values: the type of the event and a real value. The information communication mechanism using dynamic memory is relatively complex. It will not be easy for a user to change the structure of the information, type and value, transmitted in events. Anyway, it can be performed modifying the Modelica and C data structures that support the communication mechanism. In order to improve the mechanism for managing the information transmitted in events, additional information structures will be included to the DEVSLib library, e. g., giving the possibility to transmit arrays or matrices instead of only real values.

3 Introducing messages in Modelica

A conclusion of the performed work is that discrete event system modeling with Modelica, using the process-oriented approach, is not an easy task. The components required for modeling these kind of systems and the solutions proposed for the problems are relatively complex. The developed libraries provide some functionalities for discrete-event system modeling with Modelica, using the process-oriented approach. Still, there are some problems without a solution, like the one-to-many connections in DEVSLib and the polymorphism of the information transmitted at event instants.

In this section the model communication using messages in Modelica is presented. The authors also propose a possible implementation of this mechanism that will be discussed in Section 4.

3.1 Motivation

The main difficulty observed in the presented work is the model communication mechanism. This is the way models are connected and communicate.

The connection of models in Modelica is represented by the `connect` equation. In a connection equation the value of the variables at the ends of the connection are either equaled, or summed and equaled to zero. A connection between discrete-event models does not establish any relation between variables of both models, but is used to communicate some information that has been generated in one model and is transmitted to another. Both connection concepts mean different things.

Event management is also different between Modelica and DEVS discrete-event systems. An event in Modelica involves a change in the value of a boolean condition that either makes the structure of the model to change, or performs a change in the discrete time variables or the state variables of themodel. Events in DEVS discrete-event systems represent a change in the state of the system or its discrete time variables, and usually also involves the exchange of information between models. This is an instantaneous transmission/reception of an impulse of information between models at the time of an event. Event management in discrete-event systems involve additional things than in Modelica, because of this information communication.

In order to make the development of discrete-event systems more simple and easy, a new concept is proposed and introduced in Modelica. This concept is the messages communication mechanism. The messages mechanism provides the capacity for communicating impulses of information between models at event instants.

3.2 Messages and mailboxes

The model communication mechanism using messages involves two parts: the message itself and the mailbox. The message represents the information either traveling from one model to another, or inside a model itself. The mailbox receives the incoming messages and stores them until they are read. The mailbox also represents the concept of a bag of events in the Parallel DEVS formalism.

The characteristics of the model communication using messages are the following:

- A message can be sent to any available mailbox. Available mailboxes are the ones that can be referenced from the model that sends the message, either accessing directly or using a connection.
- The mailbox warns the model when new incoming messages are received.
- Once received, the message can be read from the mailbox.
- The transmission of messages between models has to be performed instantly. Any message sent from one model will be immediately received by another model.
- Messages can be received simultaneously, either in the same or different mailboxes.
- The information transported by a message, the content, is independent from the message communication mechanism. It is a task of the user to define the structure of that information using the existing components of the Modelica language, so it can be managed by the models.
- Messages can be of different types. A mailbox can store any message independently of its type. The type of the message has also to be independent from the content of the message.
- Received messages have to be stored temporarily in the mailbox, until they are read.
- Message communication has to be performed in two stages: sending and reception. The sending involves the transmission of any message in the system at a given point in time, so all the messages sent are stored in the mailbox at the end. After the sending, all the messages are available for reception in each mailbox and can be read and managed as required. If a model sends several messages to the same mailbox, all the sent messages have to be stored in the mailbox before the first message can be read by the receiver.

3.3 Message sending, transmission, detection and treatment

A message can be sent from one model to any other model that contains a mailbox, even if no connection between models is available.

Mailboxes can also be shared between models. Sharing a mailbox represent that several models can access to the message storage that it represents. Each model sharing the mailbox can access the messages stored, reading or extracting them from it. Read messages are kept in the mailbox until they are extracted, or fetched, from it.

A special case of mailboxes are the ones defined inside connectors. Two mailboxes, inside connectors, connected using a connect equation represent a bidirectional message communication pipe. They will act as input/output mailboxes instead of only receiving messages. A message sent to one end of the pipe will be transported to the opposite end, and vice-versa. If more than two models are connected to the same pipe, a copy of the message will be transported to each receiver connected to the pipe. This provides a message broadcast functionality that also emulates the event transmission in DEVS, however in DEVS the communication is not bidirectional. The connect equation functionalities in Modelica have to be extended in order to support this mailbox behavior. An example of this behavior is shown in Figure 6.

The detection of a message is implicit in the action of sending it, since they are transferred instantly. Every time a model sends a message to a mailbox, the simulator knows that the message will be received by another model and will have to be treated properly.

The treatment of each message has to be defined by the user. The mailbox warns when a new message has arrived. The mailbox activates a listener function that can be used as a condition to detect any incoming message, used with statements like `when` or `if` in Modelica. This does not mean that the new message condition has to be effectively checked at each simulation step, because it is notified by the send message operation. Once a new message arrives to a mailbox, the arrived message or messages have to be read and treated.

4 Proposal of implementation

This section contains a proposal of implementation in Modelica of the previously described message communication mechanism. This implementation is based

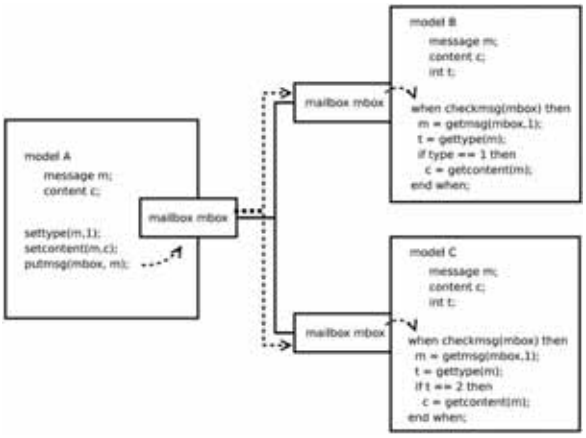

Figure 6. Model communication with messages using connectors.

on the definition of data structures that support the message and mailbox concepts, and the definition of the operations that can be performed with both data structures. Messages and mailboxes have to be defined as new predefined classes that have to be treated in a singular way, allowing objects of type message or mailbox. Due to the current Modelica language specification, the proposed implementation differs from the mechanism described above. The Modelica language will need to be extended in order to support the messages mechanism.

4.1 Data structures

There are two data structures needed to manage the messages mechanism. These are the definition of the message itself and the structure to support the mailbox that receives the defined messages.

The message structure contains two components: the type and the content. The type of a message can be represented with an integer value. It is used to separate the messages of the system in different classes. The content represents the information transported by the message. The content of a message is defined by the user and has to be independent from the message management mechanism. Thus, any mailbox can receive messages with any content and of any type. It is a task of the user to distinguish between the types of the messages and their contents. The content of the message is represented by a reference to an external data structure in C defined by the user. The user has to provide this data structure and the functions required to manage it using the reference in Modelica. Because of this definition, a message will be composed by two integer values: the type and the reference to the content.

The second structure required in the messages mechanism is the mailbox. A mailbox is a temporary storage for messages. If a message is sent to a mailbox, it is stored in the mailbox until the receiver reads it. The number of stored messages in a mailbox is not limited, so this structure has to be able to change its dimension depending on the number of stored messages. The implementation of a mailbox is very similar to the currently implemented linked-lists for storing delayed entities during processes.

4.2 Operations

The operations that can be performed with the previously described structures are defined below. Each operation is defined with its parameters and a short description of its behavior.

Mailbox operations

- `newmailbox(mailbox)`. Initializes the mailbox.
- `checkmsg(mailbox)`. Warns about the arrival of a new message. It changes its value from false to true and immediately back to false at each message arrival event.
- `newmsg()`. Detects the arrival of a message to any of the mailboxes declared in the model. This helps to manage the simultaneous arrival of messages in different mailboxes.
- `nummsg(mailbox)`. Returns the number of waiting messages stored in the mailbox.
- `readmsg(mailbox,select)`. Reads a message from the mailbox. The select parameter represents a user-defined function used to select the desired message to be read from the mailbox.
- `getmsg(mailbox,select)`. Fetches a message from the mailbox, deleting it. The select parameter is used in the same way as in the `readmsg` function.
- `putmsg(mailbox,message)`. Sends the message to the mailbox.

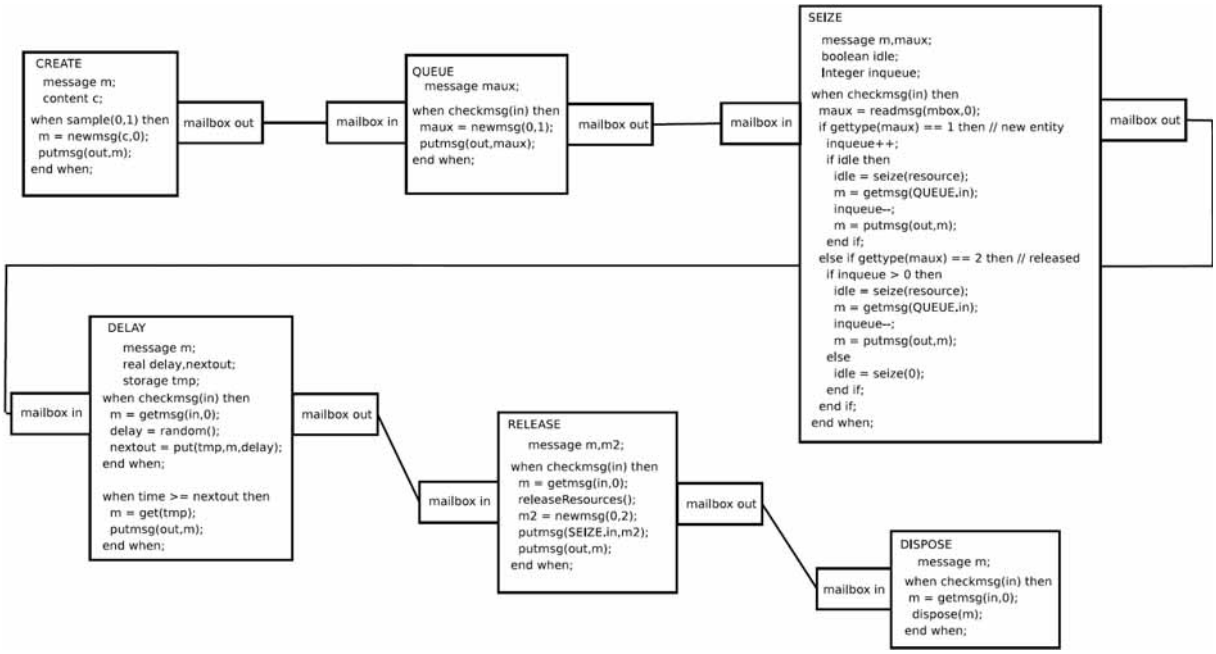

Figure 7. Example of a SIMAN single-queue system modeled using messages.

Message operations

- `newmsg(content,type)`. Creates a new message with the defined type and content.
- `gettype(message)`. Returns the type of the message.
- `settype(message,newtype)`. Updates the type of the message to the value of newtype.
- `getcontent(message)`. Reads the content of the message.
- `setcontent(message,newcontent)`. Inserts the newcontent into the message.

An example of a SIMAN single-queue system, with the Create, Queue, Seize, Delay, Release and Dispose blocks, modeled using the described messages mechanism is shown in Figure 7. Each block of the figure contains the pseudo-code that implements the basic actions for the entity management and communication. The select function, in the `readmsg` and `getmsg` functions, has been simplified and only represents the type of message to be read or extracted.

5 Conclusions

It has been observed that process-oriented modeling of discrete-event systems in Modelica is a difficult task. Several Modelica libraries have been developed to provide more discrete-event system modeling functionalities to Modelica, especially for modeling systems using the process-oriented approach. The implementation of these libraries present some problems and restrictions, and the solutions proposed and implemented are complex, hard to understand and difficult to maintain. In order to facilitate the development of discrete-event system models in Modelica, the message communication mechanism has been introduced and described. A possible implementation of this mechanism in Modelica has also been proposed.

Acknowledgments

This work has been supported by the Spanish CICYT, under DPI 2007-61068 grant, and by the IV PRICIT (Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid 2005-2008), under S-0505/DPI/0391 grant.

References

[1] Dynasym AB. *Dymola Dynamic Modeling Laboratory User's Manual*. http://www.dymola. com/ , 2006.

[2] 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.

[3] T. Beltrame, F.E. Cellier. *Quantised State System Simulation in Dymola/Modelica using the DEVS Formalism*. In Proceedings of the 5th International Modelica Conference, pages 73–82, 2006.

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

[5] M.C. D'Abreu, G.A. Wainer. *M/CD++: Modeling Continuous Systems Using Modelica and DEVS*. In Proc. 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 229–236, 2005.

[6] http://www.euclides.dia.uned.es/.

[7] J.A. Ferreira and J.P. Estima de Oliveira. *Modelling Hybrid Systems using Statecharts and Modelica*. In Proc. 7th IEEE International Conference on Emerging Technologies and Factory Automation, pages 1063–1069, 1999.

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

[9] H. Lundvall, P. Fritzson. *Modelling concurrent activities and resource sharing in Modelica*. In Proceedings of the SIMS 2003 - 44th Conference on Simulation and Modeling, 2003.

[10] W.D. Kelton, R.P. Sadowski, D.T. Sturrock. *Simulation with Arena (4th ed.)*. McGraw-Hill, Inc., New York, NY, USA, 2007.

[11] J. Kriger. Trabajo práctico 1: Antiguo reloj des pertador. http://www.sce.carleton.ca/faculty /wainer/wbgraf/samplesmain_1.htm.

[12] A.M. Law. *Simulation Modelling and Analysis (4th ed.)*. McGraw-Hill, 1221 Avenue of the Americas, New York, NY, 2007.

[13] P. L'Ecuyer. *Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators*. Oper. Res., 47(1):159–164, 1999.

[14] S.E. Mattsson, M. Otter, H. Elmqvist. *Modelica Hybrid Modeling and Efficient Simulation*. In Proc. 38th IEEE Conference on Decision and Control, pages 3502–3507, 1999.

[15] 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, pages 314–319, 1998.

[16] M. Otter, H. Elmqvist, S.E. Mattsson. *Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle*. In CACSD'99, pages 151–157, 1999.

[17] M. Otter, K.-E. Årzén, I. Dressler. *StateGraph - A Modelica Library for Hierarchical State Machines*. In Proc. 4th International Modelica Conference, pages 569–578, 2005.

[18] C.D. Pegden, R.P. Sadowski, R.E. Shannon. *Introduction to Simulation Using SIMAN*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[19] M.A. Pereira Remelhe. *Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment*. In Proc. 2nd International Modelica Conference, pages 203–207, 2002.

[20] V. Sanz, A. Urquia, S. Dormido. *ARENALib: A Modelica Library for Discrete-Event System Simulation*. In Proc. 5th International Modelica Conference, pages 539–548, 2006.

[21] V. Sanz, A. Urquia, S. Dormido. *DEVS Specification and Implementation of SIMAN Blocks Using Modelica Language*. In Proc. Winter Simulation Conference 2007, pages 2374–2374, 2007.

[22] G. Wainer. *CD++: A Toolkit to Develop DEVS Models*. Softw. Pract. Exper., 32(13):1261–1306, 2002.

[23] B.P. Zeigler, Tag Gon Kim, H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[24] B.P. Zeigler, H.S. Sarjoughian. *Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models*. http://www.acims.arizona.edu/PUBLICATIONS/.93

Corresponding author: Victorino Sanz
Dpto. Informática y Automática,
ETSII Informática, UNED
Juan del Rosal 16, 28040 Madrid, Spain
vsanz@dia.uned.es

Accepted EOOLT 2008, June 2008
Received: July 30, 2008
Revised: August 15, 2008
Accepted: August 20, 2008