Parallel DEVS and Process-Oriented Modeling in Modelica

Victorino Sanz Alfonso Urquia Sebastian Dormido
Dpto. Informática y Automática, ETSI Informática, UNED
Juan del Rosal 16, 28040, Madrid, Spain
{vsanz,aurquia,sdormido}@dia.uned.es

Abstract

This manuscript presents a new free Modelica library, named DESLib and composed of four packages: RandomLib, DEVSLib, SIMANLib and ARENALib. DESLib has been designed and implemented to facilitate the description of discrete-event models using the Parallel DEVS formalism (using DEVSLib), and to facilitate the process-oriented modeling of logistic systems (using SIMANLib and ARENALib). SIMANLib and ARENALib models are designed as DEVS models, and implemented using DEVSLib, to facilitate its development, comprehension and maintenance. RandomLib includes functionalities to generate random numbers and random variates, and facilitate the development of stochastic models. The communication mechanism used to transport information between models in DESLib is presented. This mechanism facilitates the combination of DEVS and process-oriented models to describe discrete-event systems at multiple levels. DESLib also includes interfaces to combine its components with other Modelica libraries, facilitating the composition of multi-formalism and multi-domain hybrid models. DESLib can be downloaded from `http://www.euclides.dia.uned.es`.

Keywords: discrete-event systems, hybrid modeling, Parallel DEVS, process-oriented modeling, random number generation, stochastic simulation, logistic model

1 Introduction

Modelica provides language constructs to describe the trigger conditions of time and state events, and also the actions associated to the events [1]: (1) update the value of discrete-time variables and reinitialize continuous-time state variables, using *when* clauses; and (2) change the mathematical description of equations and assignments, using the *if* statement.

These features have facilitated the development of state machine models [2, 3] and also of libraries supporting different formalisms for discrete-event system modeling. Some of these libraries are StateCharts [4], StateGraph [5], HyAuLib [6], PetriNets [7] and ExtendedPetriNets [8]. Other approach is described in [9], in which the discrete-event system is described using an external tool that generates the corresponding Modelica code. The use of Modelica language to describe discrete-event models of communication networks is presented in [10].

1.1 Parallel DEVS Formalism

The support of the Modelica language to the DEVS (Discrete EVent System specification) formalism [11] is an open research area. The feasibility of constructing basic atomic and coupled DEVS models in Modelica was demonstrated in [12]. Another implementation of this formalism was performed in the ModelicaDEVS library [13], designed to simulate continuous-time systems using the Quantized State System (QSS) integration methods [14, 15].

An atomic model is the simplest component that can be defined using Parallel DEVS (PDEVS) [16], and can be formally described with the tuple $M = (X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$, where X is the set of input ports and values, Y is the set of output ports and values, S is the set of sequential states, δ_{ext}, δ_{int} and δ_{con} are the transition functions, λ is the output function and ta is the time advance function. An atomic model updates its state with δ_{int} every time a given amount of time (defined by ta) is elapsed without any external input, thus triggering an internal event. An output can be generated, using the λ function, before executing δ_{int}. Inputs are stored in a *bag*, which is a set with possible multiple occurrences of its elements. When any input is received, the external event is triggered and the state is updated by δ_{ext}, that manages the elements in the bag. The simultaneous occurrences of an external and an internal event trigger a confluent event, and the state is updated by δ_{con}.

A coupled model can be described as a composition of other atomic or coupled models. It is specified in the PDEVS formalism with a tuple $M = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC)$, where X is the set of input ports and values, Y is the set of output ports and values, D is the set of component names, M_d is the set of DEVS components, EIC is the set of connections between input ports and components, EOC is the set of connections between components and output ports and IC is the set of internal connections between components.

1.2 Process-Oriented Modeling

According to the process-oriented "world view", systems are described from the point of view of the entities that flow through them using the available resources [17]. This modeling methodology, widely used for describing complex logistic systems, is supported by several modeling languages (e.g., GPSS/H, SLAM II, SIMSCRIPT II.5, SIMAN, SIMULA and SIMPL/1) and integrated environments (e.g., Arena, AutoMod, ProModel, Witness and SIMPROCESS).

Arena [18] is a widely used process-oriented simulation environment. It includes components to describe the flowchart diagram of the system, that represents the flow of entities. That diagram includes the processes and actions performed to the entities along the simulation run. Other components allow to describe the characteristics of those processes and actions (such as, the organization policy of the queues, the number of available resources, etc.). Components in Arena are internally described by the lower-level components of the SIMAN language [19].

Process-oriented modeling with Modelica is an attractive research field. An introduction of operations management modeling with Modelica is described in [20], where the authors present a case study of an inventory system and describe the problems encountered when modeling these kind of systems using Modelica. Almost no other work has been performed in this field.

2 The DESLib Modelica Library

The first objective of the research work presented in this manuscript is to facilitate the description of discrete-event models using the PDEVS formalism [16] and also facilitate the connection of these models to other hybrid models developed using other Modelica libraries.

The second objective of our work is to facilitate the process-oriented modeling of logistic systems using Modelica and also to facilitate the connection of these logistic system models to hybrid models developed using other Modelica libraries.

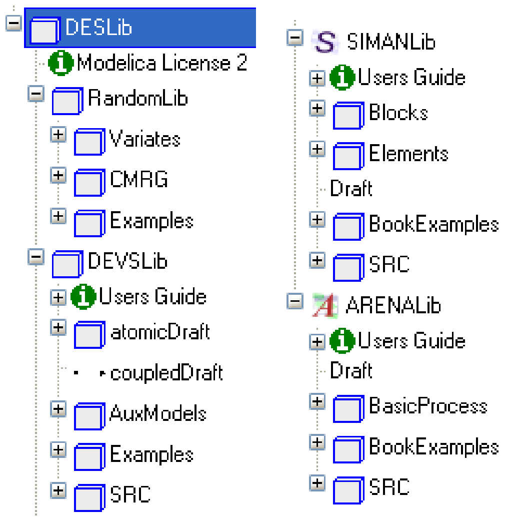


Figure 1: DESLib library.

In order to achieve these two objectives, a new Modelica library, named DESLib, has been designed and programmed using Dymola [21]. DESLib, which is freely available under the terms of the Modelica License 2, can be downloaded from `http://www.euclides.dia.uned.es/`.

DESLib is composed of the following four packages: RandomLib, DEVSLib, SIMANLib and ARENALib. The general architecture of the library is shown in Fig. 1.

- The RandomLib package includes an implementation of the CMRG random number generator, used by Arena, and some functionalities for random variates generation.
- The DEVSLib package facilitates the description of discrete-event models in Modelica following the Parallel DEVS formalism.
- The SIMANLib and ARENALib packages facilitate the description of process-oriented models. The components in these packages have been designed as atomic and coupled PDEVS models, and implemented using DEVSLib.

DEVSLib, SIMANLib and ARENALib use the same communication mechanism to transport information between models. This makes all their components compatible and can be combined to develop models at multiple description levels. The implemented communication mechanism is detailed in Section 4.1.

The organization of the manuscript is as follows. A description of the RandomLib package is included in Section 3. The architecture and design of the DEVS-

Lib package, together with its functionalities to describe Parallel DEVS models, are described in Section 4. The components and functionalities of the SIMANLib and ARENALib packages are described in Section 5. Two case studies are discussed in Section 6: the model of a restaurant constructed using SIMANLib and the model of an electronic factory constructed using ARENALib. Finally, some conclusions are given in Section 7.

3 RandomLib

Process-oriented models are usually stochastic [22]. The RandomLib package is used in conjunction with DEVSLib, SIMANLib and ARENALib to model discrete-event and process-oriented stochastic models of logistic systems.

RandomLib contains a Modelica implementation of the Combined Multiple Recursive Generator (CMRG) which is used in Arena [23, 24]. The implemented random number generator (RNG) gives the possibility of creating multiple random streams, and sub-streams, that can be considered as independent RNGs [24]. The generator period is close to 2^{191}, and can be divided into disjoint streams of length 2^{127}. At the same time, each stream can also be divided into 2^{51} adjacent sub-streams, each of length 2^{76}.

RandomLib is composed of three packages (see Fig. 1): CMRG, Variates and Examples. The CMRG package includes the implementation of the CMRG uniform random number generator. Although available in C, this generator has been implemented in Modelica in order to facilitate its use, comprehension and reutilization in other Modelica libraries.

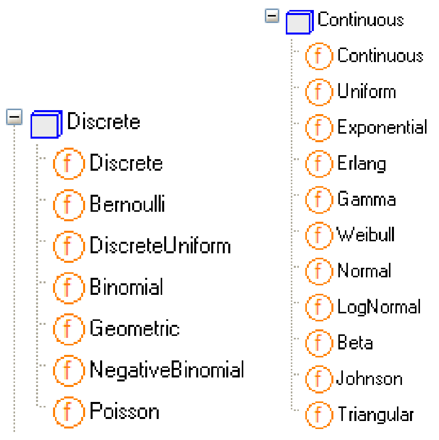


Figure 2: Discrete and continuous probability distribution functions included in RandomLib.

The Variates package includes several functions to generate random variates from continuous and discrete probability distributions. The included probability distribution functions are shown in Fig. 2. These functions use by default the CMRG generator as source of uniform random numbers. However, any other Modelica library for uniform random number generation can be used, redeclaring the record that represents the generic generator, its initialization function and the generic uniform random number generation function.

The Examples package contains several examples of random uniform and random variates generation in order to facilitate the use of the library. One of the included examples is shown in Listing. 1

```
model VariatesSimple
  "generates 5 random variates with
   Expo(5) distribution"
  Variates.Generator g "RNG";
  Real u[5] "vector of random variates";
algorithm
  // initialization of the RngStream
  when initial() then
    g := Variates.initGenerator();
  end when;
  when time <= 0 then
    for i in 1:5 loop
      // generation of variates.
      (u[i],g) :=
      Variates.Continuous.Exponential(g,5);
    end for;
  end when;
end VariatesSimple;
```

Listing 1: Random variates generation using RandomLib.

4 DEVSLib

The DEVSLib package, as shown in Fig. 1, contains the following packages and models: the UsersGuide that contains the documentation about the structure and use of DEVSLib; the atomicDraft and coupledDraft models that are used to construct new DEVS models; the AuxModels package that includes several useful auxiliary models; the Examples package that contains several examples of systems modeled using DEVSLib and; the SRC package that includes all its internal implementation and documentation.

DEVSLib supports the definition of models using the PDEVS formalism. Atomic and coupled models can be constructed with DEVSLib following their DEVS formal specification, similarly to how it is performed by other DEVS tools, such as DEVSJAVA [25], CD++ [26], or adevs [27].

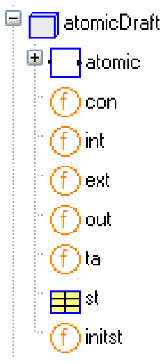


Figure 3: DEVSLib atomic model structure.

The structure of an atomic model in DEVSLib is shown in Fig. 3. The transition, output and time advance functions (*con*, *int*, *ext*, *out* and *ta* in Fig. 3) are described as Modelica functions. The state is described using a Modelica record (*st*), and initialized using the *initst* function. Any variable required to describe the state of the model can be added to that record. An example of new atomic model construction using DEVSLib is given in Section 4.3.

The development of coupled models with DEVSLib also follows its formal specification. Using the object-oriented modeling capabilities of Modelica, coupled models are constructed connecting previously developed components and including the required input/output ports.

The interconnection of the components in a coupled model may include algebraic loops. Due to the characteristics of the Modelica language, these algebraic loops are not allowed. This problem can be avoided by redefining the behavior of the models of the loop into one single atomic model, or breaking the loop inserting the *breakloop* model, included in DEVSLib, in any of its connections.

4.1 DEVSLib Model Communication

Model communication in PDEVS follows a message passing mechanism. The output function generates a new message and sends it through an output port. The message will be received, triggering an external event, by the models connected to that output port. The message may contain any kind of information, called the "value" of the message.

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

The Modelica model communication and the DEVS message passing mechanisms are conceptually different. The former equals values of variables while the latter transports information between models.

In order to allow the description of DEVS models in Modelica, a message passing mechanism has to be developed [28]. The development of this mechanism has been the most challenging problem solved during the development of the DEVSLib package. Several approaches were studied and developed, in order to implement the message passing mechanism in Modelica.

The direct implementation of the message passing mechanism in DEVSLib using Modelica connectors was studied. The mentioned DEVS implementations in Modelica [12, 13] use boolean variables inside the connectors to detect external events – i.e. received messages. However, the connectors does not allow the simultaneous reception of messages, because their variables can not be assigned with several values at the same time. Also, Modelica does not allow a variable number of objects in a model, so the message transmission can not be directly implemented.

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

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

4.2 Interface Models with Other Modelica Libraries

DEVSLib includes several interface models to translate messages into continuous-time signals, and viceversa. The use of these interface models allows to combine models developed using DEVSLib with the components of other Modelica libraries.

There are two mechanisms used in the continuous-to-discrete translation: the cross-functions and the quantization. The former translates the value of a continuous-time signal into a message every time the signal crosses a given threshold, in one direction (upwards or downwards). The models “crossUP” and “crossDOWN” implement this behavior in DEVSLib. The quantization mechanism is implemented by the “quantizer” model. This model generates a message every time the value of the continuous-time signal changes in a predefined quantum, similarly to the behavior of the QSS first-order method [15].

On the other hand, the discrete-to-continuous translation is performed generating a piecewise-constant signal whose value is the value of the last message received. The model “DICO” (DIscrete-to-COntinuous) implements this behavior in DEVSLib.

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

4.3 Model Development with DEVSLib

The development of new models using DEVSLib requires the following steps:

- Declare the input and output ports of the model.
- Define the state variables of the model and their initialization.
- Define the transition, output and time advance functions.

A “bank teller” system is described in this section as an example of model construction. In this system the customers arrive to the bank and wait their turn in the queue. If the teller is idle, the customer is served immediately. Otherwise, the teller will serve the first customer in the queue. When finished, the customer leaves the bank and the teller serves another customer if anyone else is waiting, or waits for a new arrival.

The model of this system is composed of two atomic models: the *customers* and the *teller*. The *customers* model represents the arrivals of new customers. The inter-arrival time follows an exponential probability distribution with mean 10 mins. The *teller* model represents the person serving customers and the queue. The time spent by a customer with the teller follows an exponential probability distribution with mean 8 mins.

The Parallel DEVS specification of the *customers* model is the following:

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

where:

$X_M = \emptyset$
$S = \Re_0^+$
$Y_M = \{"out", \{1\}\}$
$\delta_{int}(sigma) = interArrivalTime$
$\delta_{ext}() =$ nothing since $X_M = \emptyset$
$\delta_{con}() =$ nothing since $X_M = \emptyset$

```
model customers "Customers arrival"
  replaceable Real interarrival = 1;
  extends AtomicDEVS(numIn=1,numOut=1,
    redeclare record State = st);
  redeclare function Fint =
    int(iat=interarrival);
  redeclare function Fout = out;
  redeclare function Fta = ta;
  redeclare function initState = initst;
  Interfaces.outPort outPort1;
equation
  iEvent[1] = 0;
  // OUTPUT PORTS
  oEvent[1] = outPort1.event;
  oQueue[1] = outPort1.queue;
end customers;

record st "state of the model"
  Real sigma;
end st;

function initst "state initialization func."
  output st out;
algorithm
  // first internal transition at time = 1
  out.sigma := 1;
end initst;

function int "Internal Transition Function"
  input st s;
  input Real iat //inter-arrival time;
  output st sout;
algorithm
  sout := s;
  sout.sigma := iat;
end int;

function out "Output Function"
  input st s;
  input Integer queue[nports];
  input Integer nports;
  output Integer port[nports];
protected
  stdEvent y;
algorithm
  y.Value := 1;
  // send output event with message y
  sendEvent(queue[1],y);
  port[1] := 1;
end out;

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

Listing 2: DEVSLib code for the customers model of the Bank Teller system.

$\lambda(sigma) = 1$
$ta(sigma) = sigma$

The *interArrivalTime* is a continuous-time input of the model that represents the time between customer arrivals, similarly to the continuous-time inputs described in the DEV&DESS formalism [11].

The implementation of the *customers* model using DEVSLib is shown in Listing 2. The code has been adapted from the atomicDraft model, and follows its structure (see Fig. 3). The input port has been removed, and the iEvent array is set to 0 because the model will never receive a message. The state record contains a variable that represents the interval for the next internal transition (sigma). The model will execute its first internal transition at time 1, due to the initialization of sigma. The external and confluent transition functions (*ext* and *con*) have been removed because the model has no input ports. The internal transition function (*int*) sets the value of sigma with the input “iat”, which is a continuous-time input that represents the probability distribution for the inter-arrival times. The output function generates a new message, that represents a new customer, and sends it through the output port. The time advance function only returns the value of sigma, set by δ_{int}.

The Parallel DEVS specification of the *teller* model is the following:

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

where:

$X_M = \{\text{"in"}, \{1\}\}$
$S = \{\text{"active"}, \text{"passive"}\} \times \Re_0^+ \times \mathbb{N}$
$Y_M = \{\text{"out"}, \{1\}\}$
$$\delta_{int}(phase, sigma, nqueue) = \begin{cases} (\text{"active"}, PT, nqueue - 1) & if\ nqueue > 0 \\ (\text{"passive"}, \infty, 0) & otherwise \end{cases}$$
$$\delta_{ext}(phase, sigma, nqueue, u, e, X) = \begin{cases} (\text{"active"}, PT, nqueue) & if\ \text{"passive"} \\ (phase, sigma - e, nqueue + 1) & otherwise \end{cases}$$
$$\delta_{con}(phase, sigma, nqueue, u, e, X) = \delta_{ext}(\delta_{int}(phase, sigma, nqueue), u, 0, X)$$
$\lambda(phase, sigma, nqueue) = 1$
$ta(phase, sigma, nqueue) = sigma$

PT is a continuous-time input of the model that represents the service time for each customer.

The implementation of the *teller* model is similar to the *customers* model, and also follows the structure of the atomicDraft model. The *teller* has one input and one output ports. Its state record includes the operational mode of the teller (phase, initialized to 1 == “idle”), the interval for the next internal transition (sigma, initialized to infinity) and the queue (nqueue, initialized to 0 == “empty”). Since the arrival of a new customer does not include additional information (like its name, age, etc.), the queue only stores the number of customers waiting. At external events, when a new customer arrives it is either serviced (teller “idle” with phase == 1) or waits in queue (teller “busy” with phase == 2). The value of sigma is set with the service time (also received as a continuous-time input) or the rest of the service time of the customer being processed, if the teller is “idle” or “busy” respectively. At internal events, when the serviced customer leaves, the teller checks the value of the queue. If any other customer is in the queue, the teller starts its service and sets sigma to the new service time. If no customers are waiting, the teller becomes “idle” and waits for a new arrival setting the sigma to infinity. The output function generates a new message that represents the customer leaving the bank.

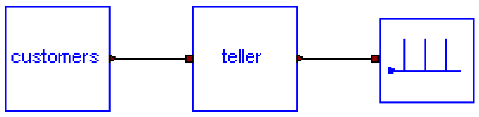


Figure 4: Bank teller system modeled using DEVSLib.

The system constructed using DEVSLib is shown in Fig. 4. It includes the code for generating the random inter-arrival and service times, using RandomLib. The model connected to the output of the *teller* only shows the departure of the customers. The results after simulating the model during 20 time units are shown in Fig. 5, containing the arrival, the departure and the number of customers in queue over the simulation. The average number of customers in queue over a long simulation (10^6 time units) is shown in Fig. 6. That

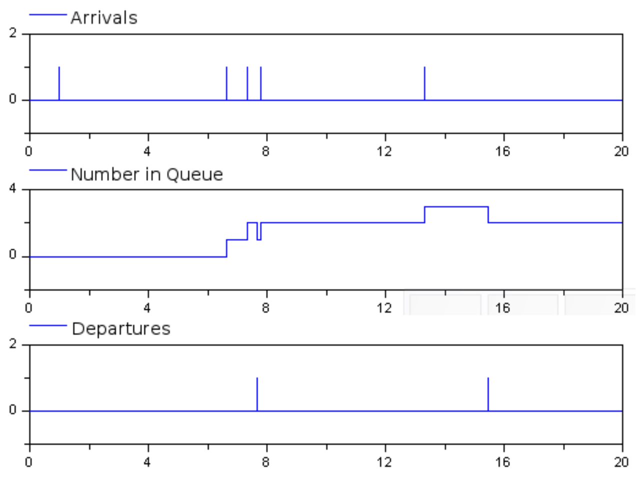


Figure 5: Bank teller system simulation results.

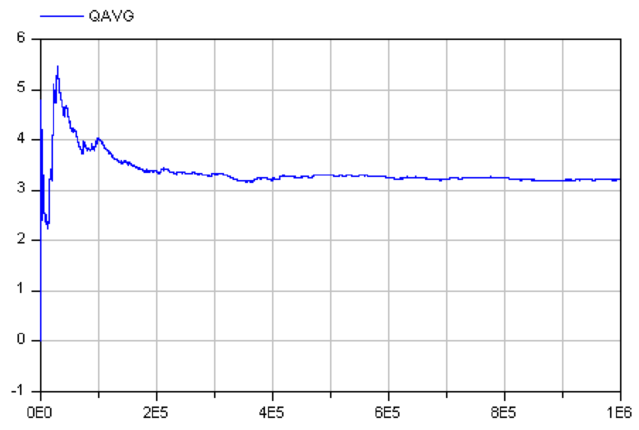


Figure 6: Average number of customers in queue from the simulation of the bank teller system.

result tends to the analytical result (an average of 3.2 jobs in queue) of the equivalent M/M/1 queue system.

5 SIMANLib and ARENALib

SIMANLib and ARENALib support the process-oriented modeling methodology, in a similar fashion to SIMAN and Arena, but with limited capabilities. Systems modeled using SIMANLib and ARENALib are composed of two parts: the flowchart diagram and the experimental data. The flowchart diagram describes the flow of entities through the system. It is defined by the *blocks* in SIMANLib and the *flowchart modules* in ARENALib. The experimental data describes the particular information of an experiment to be performed with the system. It corresponds to the amount of available resources, the characteristics of the queues, the statistical information to be recorded, etc. It is defined by the *elements* in SIMANLib and the *data modules* in ARENALib.

The structure of both packages (see Fig. 1) is similar, and are divided into two areas: the users area and the developers area. The users area in SIMANLib is composed of the Blocks (containing flowchart components), Elements (containing data components), Draft (used to construct new models) and BookExamples (containing the implementation of several examples described in [19]) packages. The users area in ARENALib is composed of the BasicProcess (containing both flowchart and data modules) and BookExamples (containing the implementation of several examples described in [18]) packages. Both, SIMANLib and ARENALib, contain a UsersGuide package that includes a description of their characteristics, structure and use. The developers area in both packages contains the SRC package, with the internal implementation of their components and the developers documentation.

The communication between flowchart components also follows a message passing mechanism, where the value of the messages are the entities. Entities are created by the Create blocks or modules, and sent to the next component. Each component performs a process (or action) to the received entity and sends it to the next component. Entities leave the system at Dispose blocks or modules.

5.1 SIMANLib Components

SIMANLib reproduces several elements of the SIMAN language [19]. The included components are shown in Fig. 7.

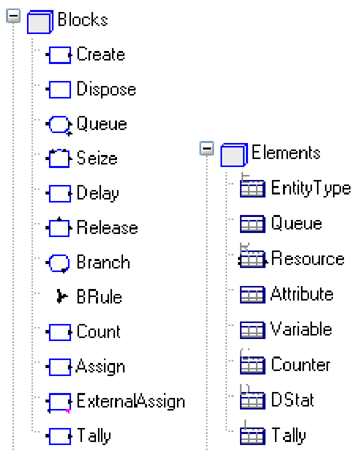


Figure 7: SIMANLib components: blocks and elements.

In order to facilitate the development and maintenance of the SIMANLib package, the SIMANLib blocks have been specified using the DEVS formalism [29] (i.e., as atomic PDEVS models) and have been implemented using the DEVSLib package. Also the Resource element has been modeled as an atomic PDEVS model, in order to manage the seize and release petitions to the resource.

5.2 ARENALib Components

ARENALib reproduces several elements of the Basic Process panel of the Arena Simulation environment [18]. The included components are shown in Fig. 8.

A previous implementation of the ARENALib package was presented in [30]. That implementation, which was directly coded in plain Modelica, was difficult to understand, maintain and modify. Arena com-

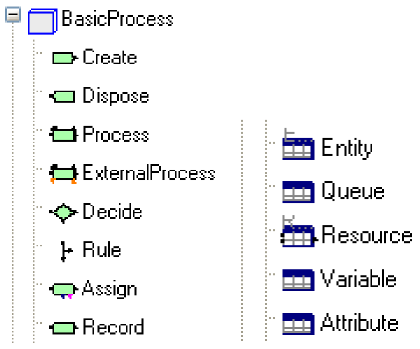


Figure 8: ARENALib components: flowchart and data modules.

ponents are developed using SIMAN constructs [18]. Analogously to SIMANLib, the components of ARENALib have been reconstructed as coupled PDEVS models, using the SIMANLib package. As an example, the internal structure of the Process flowchart module is shown in Fig. 9. The use of DEVS to describe SIMANLib and ARENALib constructs facilitates the understanding of the behavior of each library component, the development of new models and its implementation.

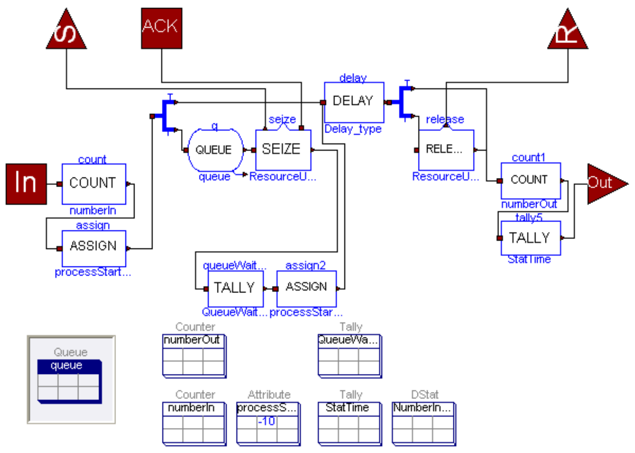


Figure 9: Internal structure of the ARENALib process module.

5.3 Hybrid System Modeling

SIMANLib and ARENALib include models that facilitate combining a process-oriented model with a continuous-time model. These models are: the External Assign block in SIMANLib and the External Process in ARENALib (which is not present in Arena). These models are shown in Fig. 10. Also, due to the compatibility between ports, any model developed using DEVSLib can be combined with SIMANLib and ARENALib, taking into account the values of the transmitted messages.

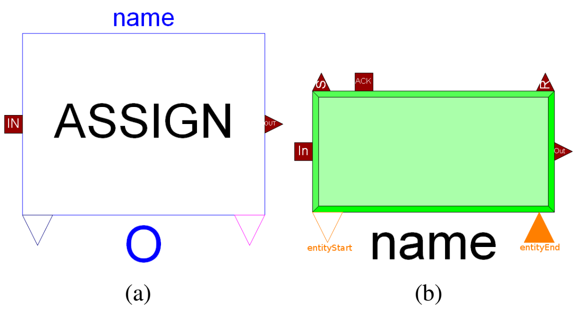


Figure 10: Hybrid system modeling components: a) SIMANLib External Assign; and b) ARENALib External Process.

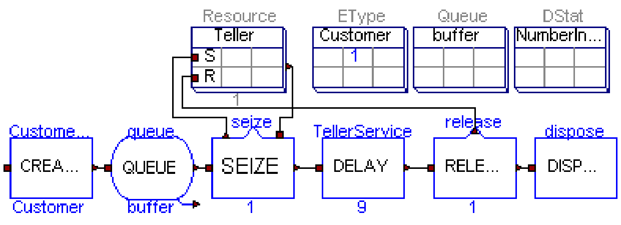
The External Assign block behaves like an Assign block, setting the value of a variable or attribute, and also includes two continuous-time output ports. These ports can be used to detect the changes in the value of the variable, and use that value in a continuous-time model.

The External Process behaves like a normal Process module, representing a process performed to the entities, but the processing time (delay) is calculated by an external continuous-time model (named “ext-process”). Every time an entity arrives to the module, and after seizing the resources, if needed, the External Process changes the value of the “entityStart” port to the reference of the received entity. The ext-process has to detect this change as a notification to start processing an entity. When the process is finished, the ext-process changes the value of the “entityEnd” port to the reference of the previously received entity. With that change, the External Process module detects the end of the process, identifies the entity and lets it continue through the flowchart diagram.

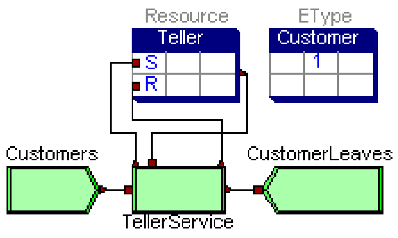
5.4 Model Development with SIMANLib and ARENALib

The “bank teller” system constructed using SIMANLib and ARENALib is presented in this section. The flowchart diagrams of both models are shown in Fig. 11

The model constructed using SIMANLib contains the following blocks: Create (that represents the arrival of customers), Queue, Seize (that together with the Release manages the availability of the teller), Delay (that represents the delay due to the service time), Release and Dispose (that represents the departure of customers). It also includes the following elements:



(a)



(b)

Figure 11: Bank teller system modeled using: a) SIMANLib; and b) ARENALib.

Resource (that represents the teller), EType (that represents the customers), Queue (that describes the organization of the queue) and DStat (that calculates the statistics for the number of customers in queue).

The model constructed using ARENALib contains the following flowchart modules: Create (that represents the arrivals of customers), Process (that represents a customer serviced by the teller) and Dispose (that represents the departure of customers). The data modules included are: Resource (representing the teller) and Entity (representing the customers).

Both models are equivalent, but as shown in Fig. 11 the components in SIMANLib perform simpler actions and more components are required to model the same behavior.

Both systems have been simulated for 10^6 time units to study the steady-state behavior. The statistical indicators are automatically calculated during the simulation run by the DStat elements. The results are shown in Table 1, including the average number of customers in queue and the half-width intervals calculated by Arena (SIMAN and Arena obtain the same results because in both cases the same seed is used to initialize the RNG).

6 Case Studies

Two case studies are presented to show the functionalities of SIMANLib and ARENALib: a restaurant and an electronic factory. The first model is analyzed running independent terminating simulations and the second one performing one long (steady-state) simulation. The results are equivalent to the ones obtained using SIMAN and Arena, respectively.

Table 1: Bank teller system simulation results using SIMANLib, ARENALib, Arena and SIMAN.

Model	Avg. Customers in Queue	Half-Width
SIMANLib	3.3073	-
ARENALib	3.1212	-
Arena	3.2089	0.22
SIMAN	3.2089	0.22

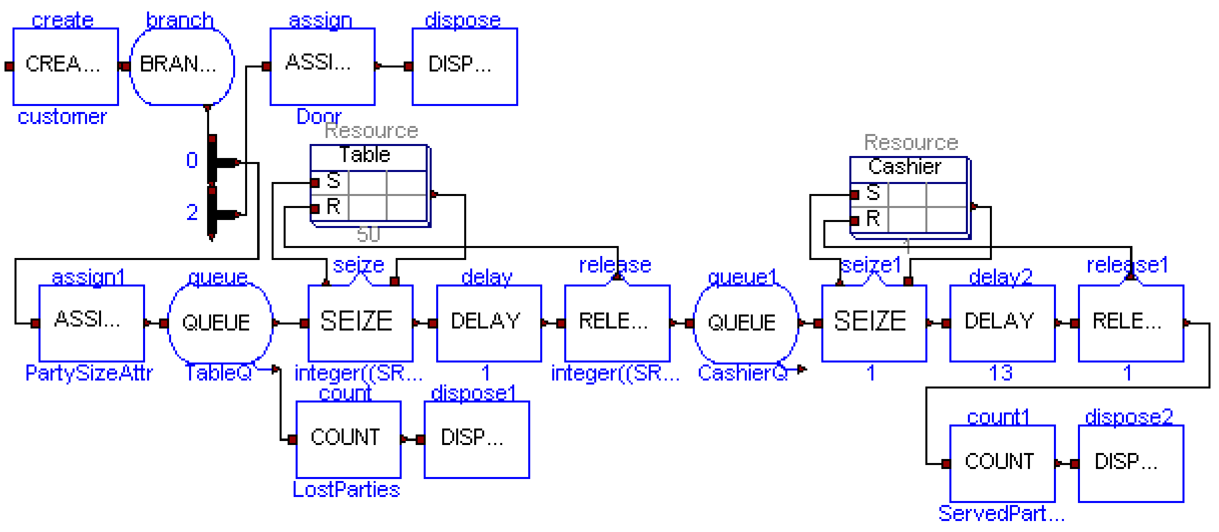
6.1 Restaurant Model

The restaurant model described in [19] has been composed using SIMANLib (see Fig. 12a). Customers arrive in groups from 2 to 5 persons and wait for an available table. If there are already 5 groups waiting, the new group leaves without waiting. The restaurant has 50 tables. Each table is for two people, so several tables may be needed for each group. When seated, the group is served and eats. At the end, the group pays the check to the cashier and leaves. The restaurant receives customers from 5 p.m. to 9 p.m., and, after that, waits until all the customers leave.

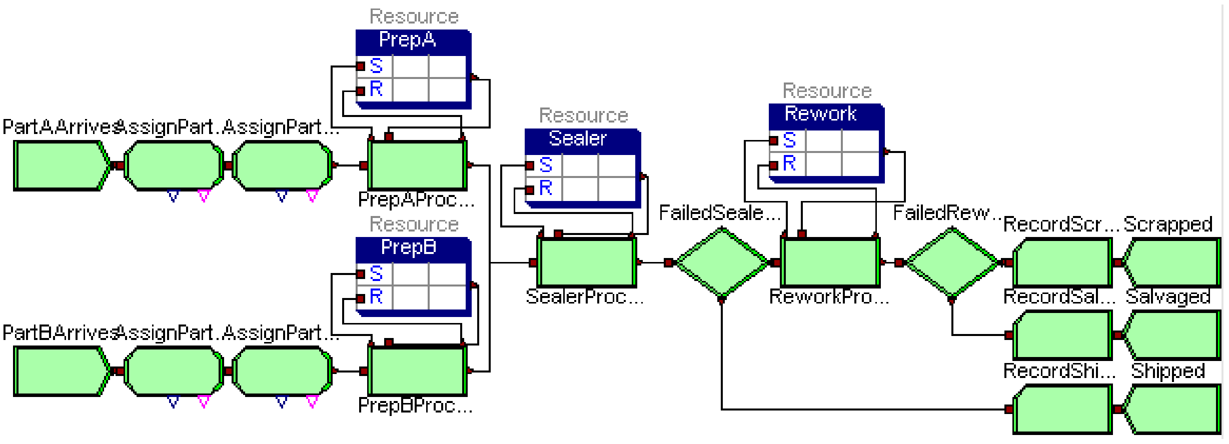
Table 2: Restaurant simulation results, comparing SIMANLib and SIMAN.

Indicator (avg.)	SIMANLib	SIMAN
groups served	136.13	135.43
groups lost	15.83	14.00
busy tables	24.49	24.25
groups waiting	0.62	0.72
cashier util.(%)	42.51	41.94

In order to analyze the system, 30 independent simulation runs, each of 480 time units, have been performed. Each run record statistics about the number of customers served, the number of busy tables, the number of waiting customers, the number of groups that left without entering and the utilization of the cashier. The simulation results, comparing the SIMANLib and SIMAN models are shown in Table 2 (average values).



(a)



(b)

Figure 12: Case studies: a) restaurant modeled using SIMANLib; and b) electronic assembly system modeled using ARENALib.

6.2 Electronic Factory Model

The electronic assembly and test system described in [18] has been composed using ARENALib (see Fig. 12b). Two types of electronic parts (A and B) are received in the system, are pre-processed and sealed. Each type has a different pre-processing and sealing time. After that, the sealed parts are inspected. Correct parts are shipped, and the rest need to be reworked. After the rework process, they are inspected again and classified into salvaged and scrapped.

The system has been simulated during 50000 time units, in order to evaluate its steady-state behavior. Multiple statistical indicators are automatically calculated by ARENALib. Some of these indicators (average values) are shown in Table 3 and the are compared with the results obtained with Arena, including the half-width (H-W) interval.

Table 3: Electronic factory simulation results, comparing ARENALib and Arena.

Indicator (avg.)	ARENALib	Arena	H-W
Shipped	18.650	19.774	2.273
Salvaged	89.348	81.522	8.715
Scrapped	88.564	78.125	(Insuf)
Sealer.WaitTime	0.447	0.453	0.035
Sealer.ProcessTime	2.609	2.617	(Corr)
Sealer.Utilization	0.601	0.605	0.011
Sealer.NumberInQueue	0.103	0.105	0.007
Rework.WaitTime	40.272	32.974	8.020
Rework.ProcessTime	30.033	28.452	1.823
Rework.Utilization	0.622	0.583	0.043
Rework.NumberInQueue	0.834	0.675	0.180

7 Conclusions

A new free Modelica library, named DESLib, has been designed and programmed to facilitate the development of discrete-event and process-oriented models. The library is composed of four packages: RandomLib, DEVSLib, SIMANLib and ARENALib. DEVSLib supports the development of discrete-event models following the Parallel DEVS formalism. SIMANLib and ARENALib facilitate the development of process-oriented models of logistic systems, with functionalities similar to the SIMAN language and the Arena simulation environment. RandomLib contains an implementation of the CMRG random number generator, used in Arena, that combined with the other packages of DESLib facilitates the development of stochastic discrete-event models.

The hierarchic description of components in DESLib (SIMANLib compopnents constructed using DESLib, and ARENALib components constructed using SIMANLib) and the use of the DEVS formalism simplifies its understanding, maintenance, reuse and further development.

The communication mechanism developed and included in DESLib allows to transport structured information between models. This mechanism can also be easily adapted to other applications and libraries.

The description and study of stochastic discrete-event and logistic models with Modelica is supported by DESLib. Due to the interface models included in the library, the combination of DESLib with other Modelica libraries facilitates the description of complex hybrid models.

References

[1] Modelica Association. Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Language Specification (v. 3.1). Available at http://www.modelica.org/documents, 2009.

[2] Mattsson S. E, Otter M, Elmqvist H. Modelica Hybrid Modeling and Efficient Simulation. In Proc. of the 38th IEEE Conf. on Decision and Control, pp. 3502–3507, 1999.

[3] Otter M, Elmqvist H, Mattsson S. E. Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle. In Proc. of the 10th IEEE Intl. Symposium on Computer Aided Control System Design, pp. 151–157, 1999.

[4] Ferreira J, de Oliveira J. E. Modelling Hybrid Systems Using Statecharts and Modelica. In Proc. of the 7th IEEE Intl. Conf. on Emerging Technologies and Factory Automation, pp. 1063–1069, 1999.

[5] Otter M, Årzén K.-E, Dressler I. State-Graph - a Modelica Library for Hierarchical State Machines. In Proc. of the 4th Intl. Modelica Conf., pp. 569–578, 2005.

[6] Pulecchi T, Casella F. HyAuLib: Modelling Hybrid Automata in Modelica. In Proc. of the 6th Intl. Modelica Conf., pp. 239–246, 2008.

[7] Mosterman P. J, Otter M, Elmqvist H. Modelling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica. In Proc. of the Summer Computer Simulation Conf., pp. 314–319, 1998.

[8] Fabricius S. M. O. Extensions to the Petri Net Library in Modelica. ETH Zurich, Switzerland, 2001.

[9] Remelhe M. A. P. Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment. In Proc. of the 2nd Intl. Modelica Conf., pp. 203–207, 2002.

[10] Färnqvist D, Strandemar K, Johansson K. H, Hespanha J. P. Hybrid Modeling of Communication Networks Using Modelica. In Proc. of the 2nd Intl. Modelica Conf., pp. 209–213, 2002.

[11] Zeigler B. P, Kim T. G, Prähofer H. Theory of Modeling and Simulation. Academic Press, 2000.

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

[13] Beltrame T, Cellier F. E. Quantised State System Simulation in Dymola/Modelica Using the DEVS Formalism. In Proc. of the 5th Intl. Modelica Conf., pp. 73–82, 2006.

[14] Cellier F. E, Kofman E. Continuous System Simulation. Springer, 2006.

[15] Kofman E. Discrete Event Simulation of Hybrid Systems. SIAM Journal on Scientific Computing, 25(5):1771–1797, 2004.

[16] Chow A. C. H. Parallel DEVS: a Parallel, Hierarchical, Modular Modeling Formalism and its Distributed Simulator. Trans. of the Society for Computer Simulation Intl., 13(2):55–67, 1996.

[17] Derrick E. J, Balci O, Nance R. E. A comparison of selected conceptual frameworks for simulation modeling. In Proc. of the 1989 Winter Simulation Conf., pp. 711–718, 1989.

[18] Kelton W. D, Sadowski R. P, Sturrock D. T. Simulation with Arena. McGraw-Hill, 4^{th} ed., 2007.

[19] Pegden C. D, Sadowski R. P, Shannon R. E. Introduction to Simulation Using SIMAN. McGraw-Hill, 1995.

[20] Mikler J, Engelson V. Simulation for Operation Management: Object Oriented Approach using Modelica. In Proc. of the 3^{rd} Intl. Modelica Conf., pp. 207–214, 2003.

[21] Dynasim AB. Dymola Dynamic Modeling Laboratory User's Manual. http://www.dymola.com, 2009.

[22] Law A. M. Simulation Modelling and Analysis. McGraw-Hill, 4^{th} ed., 2007.

[23] L'Ecuyer P. Software for Uniform Random Number Generation: Distinguishing the Good and the Bad. In Proc. of the 33^{rd} Conf. on Winter Simulation, pp. 95–105, 2001.

[24] L'Ecuyer P, Simard R, Chen E. J, Kelton W. D. An Object-Oriented Random-Number Package With Many Long Streams and Substreams. Oper. Res., 50 (6):1073–1075, 2002.

[25] Zeigler B. P, Sarjoughian H. S. Introduction to DEVS Modeling & Simulation With JAVA: Developing Component Based Simulation Models. Available at http://www.acims.arizona.edu/PUBLICATIONS/, 2003.

[26] Wainer G. CD++: A Toolkit to Develop DEVS Models. Software: Practice and Experience, 32(13):1261–1306, 2002.

[27] Nutaro J. ADEVS - A Discrete Event System Simulator. Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson. Available at http://www.ece.arizona.edu/˜nutaro/index.php, 1999

[28] Sanz V, Urquia A, Dormido S. Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling. In Proc. of 2^{nd} Intl. Workshop on Equation-Based Object-Oriented Languages and Tools, pp. 83-93, 2008.

[29] Sanz V, Urquia A, Dormido S. DEVS Specification and Implementation of SIMAN Blocks Using Modelica Language. In Proc. of the Winter Simulation Conf., pp. 2374–2374, 2007.

[30] Sanz V, Urquia A, Dormido S. ARENALib: A Modelica library for Discrete-Event System Simulation. In Proc. of the 5^{th} Intl. Modelica Conf., pp. 539–548, 2006.