SoftwareX 23 (2023) 101438

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

MSGLib: A Modelica library for message passing communication

Victorino Sanz *, Alfonso Urquia

Dpto. de Informática y Automática, ETSI Informática, Universidad Nacional de Educación a Distancia (UNED), Spain

ARTICLE INFO

Article history:
Received 12 April 2023
Received in revised form 31 May 2023
Accepted 9 June 2023

Keywords:
Message passing communication
Modelica

ABSTRACT

MSGLib is a Modelica library designed and developed to support message passing communication and the management of data structures stored in dynamic memory. The functionality of the library facilitates the description of discrete-event models and their combination with other Modelica functionality. MSGLib has been used as a base to develop other Modelica libraries such as DEVSLib, ARENALib and ABMLib. A new version of MSGLib is presented in this manuscript, that includes user documentation, performance optimization and illustrative examples. The library has been developed and tested under Dymola and OpenModelica, and is freely distributed under the LGPL-3.0 license.

Code metadata

Current code version	v2.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00197
Code Ocean compute capsule	
Legal Code License	LGPL-3.0 or later
Code versioning system used	git
Software code languages, tools, and services used	C, Modelica, Dymola, OpenModelica
Compilation requirements, operating environments & dependencies	Modelica Standard Library 4.0.0
If available Link to developer documentation/manual	
Support email for questions	vsanz@dia.uned.es

1. Motivation and significance

Modelica is an object-oriented modeling language designed to describe mathematical models using differential and algebraic equations, and events [1]. It is widely used in academia and industry to describe models in multiple domains (e.g., electrical, power, automation, etc.). Modelica models can be described behaviouraly, by defining the equations that describe the dynamics of the system, and structurally, as a set of interconnected components. Modelica tools automatically analyze and manipulate the structure and equations of the model, and generate an executable code for the simulation [2].

Component connections for structurally defined Modelica models follow an equation-based physical rationale [3]. The description of model communications in discrete-event approaches (e.g., DEVS) follows a message-passing rationale (i.e., instantaneous communication of impulses of information). DEVS models can communicate by scheduling the generation of output messages, that are immediately transmitted to other models as external inputs [4]. In order to facilitate the description of these discrete-event systems using the Modelica language, and combine them with the rest of the language functionality, a message-passing communication (MPC) mechanism is required. The MSGLib library has been designed and developed to provide such MPC functionality. Additionally, MSGLib provides functionality to manage data structures stored in dynamic memory.

In this manuscript, the version 2.0 of the MSGLib library is presented. This new version extends the previous version by including detailed documentation of its functionality and use, performance improvements and a detailed set of illustrative examples.

2. Software description

The MSGLib library is composed of two parts: one written in Modelica, named `MSGLib.mo`, that provides the required modeling functionality; and another written in C, named `msglib.c`, that constitutes the actual implementation of the data structures and the communication mechanism. The interactions between

* Correspondence to: Juan del Rosal, 16, 28040, Madrid, Spain.
E-mail addresses: vsanz@dia.uned.es (Victorino Sanz), aurquia@dia.uned.es (Alfonso Urquia).

https://doi.org/10.1016/j.softx.2023.101438

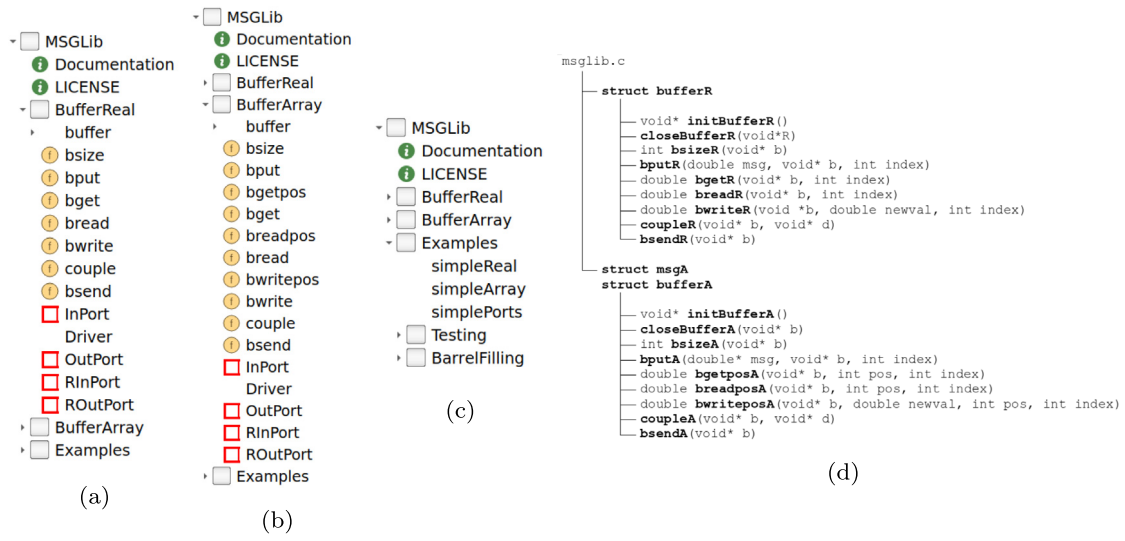

Fig. 1. Structure of MSGLib: (a) `MSGLib.mo` Documentation and `BufferReal` package, (b) `MSGLib.mo` `BufferArray` package, (c) `MSGLib.mo` `Examples` package, and (d) `msglib.c`.

both parts are described using the Modelica external function interface.

The structure of MSGLib is shown in Fig. 1. `MSGLib.mo` includes, additionally to the `Documentation` package and the `LICENSE` description, three packages:

- `BufferReal`, that includes components to describe MPC with messages of `Real` type.
- `BufferArray`, analogous to the previous package but supporting MPC using messages of `Real[]` type (i.e., array of `Real`, with each element of the array considered as a *position*).
- `Examples`, that includes multiple illustrative examples and test cases of discrete-event and hybrid systems. The `Testing` package includes test cases in combination with other Modelica functionality (e.g., algorithm and equation sections, `reinit`, and time and state events), and the `BarrelFilling` package includes the models used to describe a barrel filling facility as a hybrid system, and the implementation of a similar system described in [4].

As shown in Figs. 1(a) and 1(b), the names of the components in the `BufferReal` and `BufferArray` packages are almost identical since they provide the same functionality, to be used with different types of messages. The use of either message type can be selected by importing one of both packages in a model. Message types need to be statically defined in order to comply with the type mappings [3] between Modelica and C (cf. Fig. 1(d)). Note that messages of `Real` type can be also described using an array of length 1 (e.g., `Real[1]`) using the `BufferArray` package, but we decided to include both packages to avoid the use of array declarations for messages of `Real` type and because the `BufferReal` package, and its corresponding C code in `msglib.c`, provides a better performance in this case. The supported message types provide a wide modeling versatility, however, if custom message types are required (e.g., using a `record` class) the library can be easily adapted to support them by duplicating and modifying the existing code.

2.1. Buffers

The main component of the library is the `buffer`, which represents a container for messages. Buffers can be used as temporary storage for messages and, when coupled using the `couple` function, as the ends of a communication channel (e.g., origin and destination). Buffers are implemented as external objects in Modelica, with its corresponding constructor, and destructor functions. In C, they are described as a `struct` that stores an array of messages and an array of destinations, used to store the communication channels between buffers. Both arrays are dynamically stored in memory, depending on the requirements of the simulation. In this new version of the library, the array of messages in `BufferArray` buffers has been implemented as a double linked list in order to improve the performance of insertions and extractions of messages. The constructor and destructor C functions for the buffers are `initBuffer` and `closeBuffer`, respectively. The constructor allocates an initial memory space for the messages and initializes all the variables to represent an empty buffer. The destructor frees the memory used for the buffer.

The library includes multiple utility functions to manage messages and buffers (as shown in Fig. 1 each function in `MSGLib.mo` has its corresponding implementation in `msglib.c`, for the two supported type messages):

- `bput`, used to insert a message into a buffer.
- `bget`, used to extract a message from a buffer, and `bgetpos`, used to extract a message and return a given position of that message.
- `bread`, used to observe a message in a buffer, and `breadpos`, used to observe a given position of a message.
- `bwrite`, used to set a new value for a message in a buffer, and `bwritepos`, used to only modify a given position of a message.
- `bsize`, used to observe the number of messages in a buffer.

The library includes the `bsend` function that can be used to transmit the messages stored in a buffer to all its previously coupled destinations. Messages are directly routed to their final

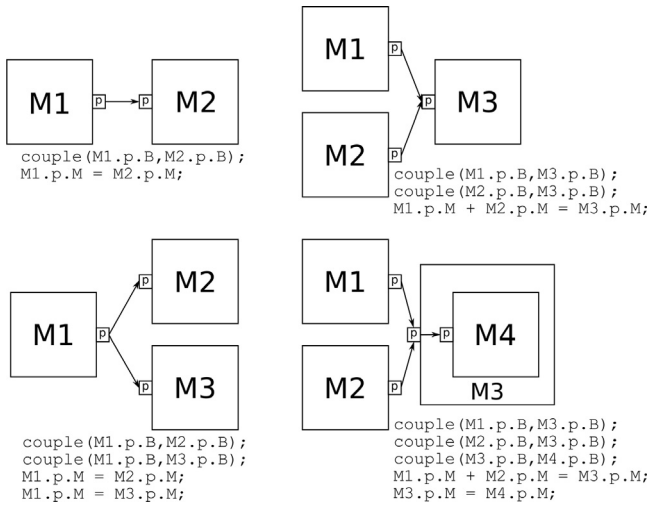

Fig. 2. Examples of port connections [5].

destinations, avoiding to store messages in intermediate buffers in the channel (e.g., having `couple(a,b)` and `couple(b,c)`, `bsend(a)` will send messages directly from `a` to `c`). This behavior provides an automatic flattening of the structure formed by the components of the model and their interconnections.

2.2. Interface for MPC

The library also includes specific models, named *ports*, to describe the MPC interface of the models. All ports include two common components: a buffer, named `B`, used to store the transmitted messages; and a Real variable, named `M`, used to synchronize the transmission of messages (cf. [5]).

The following ports are included:

- `InPort`: represents an input port used to receive messages. Additionally to the common components, input ports include a boolean flag, named `rcv`, that signals the reception of new messages and can be used as an event condition to manage the received messages.
- `OutPort`: represents an output port for outgoing messages. Additionally to the common components, output ports include an array of `Driver` models. The driver model is used to schedule and synchronize the transmission of the output messages stored in the port to its destinations (cf. [5] for a detailed description). Output messages need to be inserted in the buffer of the output port before the transmission is scheduled. The array of drivers is used to support multiple sequential transmissions during the same time instant (e.g., due to event iterations, a loop in the communications, etc.). A boolean flag, named `snd`, is used to signal the successive transmissions using different drivers of the array.
- `RInPort`: is used to define *router* input ports. Router ports are intermediate ports in a communication channel and usually describe the interface of models structurally defined. Router input ports have to be connected to input ports of any internal component of the model.
- `ROutPort`: is used to define *router* output ports. Analogous to the router input ports but need to be connected to output ports of internal components.

Connections between ports are defined with the following two actions:

- Coupling their buffers, using the `couple` function within an initial algorithm section in order to guarantee that it is only executed once in the simulation.
- Setting the value of their `M` variables within an equation section. The `M` variable of an input port has to be equaled to the sum of the `M` variables of its connected output ports.

Examples of multiple connection topologies are shown in Fig. 2. Port connections need to be textually specified in the models (e.g., the code shown in Fig. 2), since Modelica only supports the graphical description of connections between connectors.

3. Illustrative examples

Three examples are described: two very simple examples are used to illustrate the basic functionality of the library, store messages and MPC; and a more complex model describes a communication loop between three models with multiple messages transmission during the same time instant. All these examples are included in the `Examples` package of the library (cf. `SimpleReal`, `SimplePort` and `Testing.loopModular` models).

3.1. SimpleReal

This simple model illustrates the use of buffers as storage for messages. A reduced version of the code is shown in Listing 1 (prints to simulation log have been removed to simplify the code and facilitate its comprehension). When the condition `time >= 1` becomes true, two messages are inserted in the buffer. After that, the messages are sequentially read (assigned to variables `s1` and `s2`) and extracted. When the condition `time >= 2` becomes true, two new messages are inserted and extracted from the buffer.

Listing 1: Modelica code for the `SimpleReal` model.

```
model simpleReal
  import MSGLib.BufferReal.*;
  buffer b=buffer();
  Real out,x[2],s1,s2;
algorithm
  when time >= 1 then
// First event at time >= 1
    bput(1, b);
// two new messages inserted in the buffer
    bput(2, b);
    s1 := bread(b, 1);
// read contents of the buffer
    s2 := bread(b, 2);
    out := bget(b);
// extract last message from the buffer
    x[1] := out;
    s1 := bread(b, 1);
// read contents of the buffer
    s2 := bread(b, 2);
// 0 returned for missing messages
    out := bget(b);
// extract last message from the buffer
    x[2] := out;
    s1 := bread(b, 1);
// read contents of the buffer
    s2 := bread(b, 2);
// 0 returned for missing messages
  end when;
  when time >= 2 then
// Second event at time >= 2
    bput(1, b);
    bput(2, b);
    x[1] := bget(b,1);
// extract first message from the buffer
    x[2] := bget(b,1);
  end when;
end simpleReal;
```

3.2. SimplePort

This model represents a simple connection and message transmission between two ports. The code of the model, also reduced by removing prints, is shown in Listing 2. The model includes an output port, `Origin`, and an input port, `Destination`. The coupling between `Origin` and `Destination` is described using the `couple` function and the equation between their `M` variables. Messages are periodically transmitted from `Origin` to `Destination`, with an interval of 1s. The value of the message is `time+0.2`, which is inserted in the `Origin.B` buffer before scheduling the immediate transmission (`Origin.tSend` := `time` and `Origin.snd` := `not Origin.snd`). Messages are instantaneously received at `Destination`, where they are extracted from `Destination.B`.

Listing 2: Modelica code for the `SimplePort` model.

```
model simplePorts
  import MSGLib.BufferReal.*;
  OutPort Origin;
  InPort Destination;
  Real msg;
initial algorithm
    couple(Origin.B, Destination.B);
equation
  Origin.M = Destination.M;
algorithm
  when sample(0, 1) then
// GENERATION
    bput(time+0.2,Origin.B);
// message insertion
    Origin.tSend :=time;
// transmission time
    Origin.snd := not pre(Origin.snd);
    // transmission flag
  end when;
algorithm
  when Destination.rcv then
// RECEPTION
    msg := bget(Destination.B);
// extract received message
  end when;
end simplePorts;
```

3.3. LoopModular

This model is presented to illustrate the functionality included in MSGLib to describe sequential transmissions of messages at the same time instant. This behavior is commonly found in discrete-event models of cyber–physical systems (e.g., the management of collisions in computer networks). The `LoopModular` model is composed of three components of `M` type, named `m1`, `m2` and `m3`, connected in a ring topology (cf. Listing 3). Model components include a continuous-time variable, `x`, whose derivative is set to a constant value, `derx`, in order to illustrate the combination of MPC and continuous-time dynamics in a model.

Message transmissions are periodically started by `m1`, sending a message to `m2`. When a component receives a message, it extracts it from the buffer, increases a counter and, depending on the current value of the counter, it performs one of the following actions:

- If the value of the counter has not reached a defined maximum (`maxcount`), the message is forwarded to the next component in the ring.
- Otherwise, the value of `x` is reinit with the value of the message and the counter is reset to 0. Thus, the transmissions are interrupted.

Since `maxcount` is initially set to 4, the first message created by `m1` will have to perform three loops to the ring of models during the same time instant. After this first message, the amount of loops to the ring depends on the values of the counters in the other components since only the counter of the model that reaches the maximum is reset to 0 and the transmissions are always started by `m1`.

Listing 3: Modelica code for the `M` and `LoopModular` models.

```
model M
    import MSGLib.BufferReal.*;
    parameter String name;
    parameter Real derx;
    parameter Boolean start = true;
    parameter Integer maxcount = 4;
    InPort IN;
    OutPort OUT(nDrivers=maxcount);
    Integer count(start=0);
    Real x, msg;

  algorithm
    when sample(1,1) and start then
    // GENERATION
      count := pre(count)+1;
      bput(pre(x),OUT.B);
      OUT.tSend := time;
      OUT.snd := not pre(OUT.snd);
    end when;

    when IN.rcv then
    // RECEPTION
      if bsize(IN.B) > 0 then
        msg := bget(IN.B);
        count := pre(count) +1;
        if count < maxcount then
          bput(pre(x),OUT.B);
          OUT.tSend := time;
          OUT.snd := not pre(OUT.snd);
        else
          reinit(x,msg);
          count := 0;
        end if;
      end if;
    end when;

  equation
    der(x) = derx;
end M;

model LoopModular
    import MSGLib.BufferReal.*;
    M m1(derx=1, name="M1");
    M m2(start=false, derx=2, name="M2");
    M m3(start=false, derx=3, name="M3");

  initial algorithm
    couple(m1.OUT.B,m2.IN.B);
    couple(m2.OUT.B,m3.IN.B);
    couple(m3.OUT.B,m1.IN.B);

  equation
    m1.OUT.M = m2.IN.M;
    m2.OUT.M = m3.IN.M;
    m3.OUT.M = m1.IN.M;
end LoopModular;
```

4. Impact

As previously mentioned, MSGLib provides functionality to facilitate the use of a MPC approach within Modelica models. Buffers can also be used as data structures stored in dynamic memory to describe variables that may change their size during the simulation runs (e.g., a queue). This functionality has been used to facilitate the description in Modelica of models using the Parallel DEVS formalism [6], a process-oriented modeling approach analogous to Arena [7], and their application to the description of cyber–physical systems [5]. Also, an agent-based modeling approach, where agents are described as messages that flow across processes arranged in a flowchart diagram, has been proposed by the authors [8,9].

Introducing the MPC in Modelica and using it as a base to support multiple discrete-event modeling functionality enhances the versatility and applicability of the language. The description of large and complex systems, that are usually composed of a combination of heterogeneous parts, is facilitated by the application of multi-formalism and multi-level modeling approaches.

5. Conclusions

The MSGLib library extends current Modelica functionality and facilitates the use of message-passing communication to describe interactions between model components. The library can be combined with the rest of the language functionality to provide a wider modeling functionality for multi-formalism modeling. The presented version of the MSGLib library includes detailed user documentation, code optimizations designed to improve the performance of the simulations by reducing the operations required to manage the buffer data structures, and multiple illustrative examples aimed to facilitate the use of the library and to serve as a base for other modeling applications. It is freely distributed under the LGPL license and can be downloaded from https://github.com/vsanzp/MSGLib.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This research was supported by Vicerrectorado de Investigación, Transferencia de Conocimiento *y* Divulgación Científica of Universidad Nacional de Educación a Distancia (UNED), Spain [grant "Convocatoria Proyectos de Investigación UNED 2022"].

Data availability

Data will be made available on request

References

[1] Fritzson P. Principles of object-oriented modeling and simulation with Modelica 3.3: A cyber-physical approach. Wiley-IEEE Computer Society; 2014.
[2] Cellier FE, Kofman E. Continuous System Simulation. Secaucus, NJ, USA: Springer-Verlag New York, Inc. 2006.
[3] Modelica association. 2021, [Modelica] – A unified object-oriented language for systems modeling. Language spec. v. 3.5, Accessed 2023. https://modelica.org/documents/MLS.pdf.
[4] Zeigler BP, Muzy A, Kofman E. Theory of modeling and simulation: Discrete event and iterative system computational foundations. 3rd ed.. New York: Academic Press; 2018.
[5] Sanz V, Urquia A. Cyber-physical system modeling with Modelica using message passing communication. Simul Model Pract Theory 2022;117:102501. http://dx.doi.org/10.1016/j.simpat.2022.102501.
[6] Sanz V, Urquia A, Cellier FE, Dormido S. System modeling using the Parallel DEVS formalism and the Modelica language. Simul Model Pract Theory 2010;18(7):998–1018. http://dx.doi.org/10.1016/j.simpat.2010.03.004.
[7] Sanz V, Urquia A, Cellier FE, Dormido S. Hybrid system modeling using the SIMANLib and ARENALib Modelica libraries. Simul Model Pract Theory 2013;37:1–17. http://dx.doi.org/10.1016/j.simpat.2013.05.005.
[8] Sanz V, Bergero F, Urquia A. An approach to agent-based modeling with Modelica. Simul Model Pract Theory 2018;83:65–74. http://dx.doi.org/10.1016/j.simpat.2017.12.012.
[9] Sanz V, Urquia A. Combining PDEVS and Modelica for describing agent-based models. Simulation 2023;99(5):455–74. http://dx.doi.org/10.1177/00375497221094873.