DEVS GRAPH IN MODELICA FOR REAL-TIME SIMULATION

Alfonso Urquia, Carla Martin-Villalba
Departamento de Informática y Automática
Universidad Nacional de Educación a Distancia (UNED)
Juan del Rosal 16, 28040, Madrid, Spain
Email: {aurquia,carla}@dia.uned.es

Mohammad Moallemi, Gabriel A. Wainer
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, Canada
Email: gwainer@sce.carleton.ca

KEYWORDS

Modelica, DEVS, DEVS Graph, real-time simulation

ABSTRACT

Two new Modelica libraries are presented. The first library, named GGADLib, supports the DEVS graph notation in Modelica. The second Modelica library, named UDPLib, allows sending and receiving data using the User Datagram Protocol (UDP). GGADLib uses UDPLib. As a result, DEVS graph models composed using GGADLib can receive and send data (input and output events, according to DEVS terminology) using UDP. This feature allows Modelica DEVS graph models to communicate with hardware and with other models, e.g., with models developed using other languages and tools, and running in other computers. The implementation of UDPLib and GGADLib is discussed in this paper. Their use is illustrated by means of a case study: evaluating an obstacle avoidance controller for the e-puck mobile robot. The UDPLib and GGADLib Modelica libraries can be freely downloaded from http://www.euclides.dia.uned.es/

INTRODUCTION

The object-oriented modeling language Modelica facilitates describing the type of models commonly used in control engineering (Åström et al., 1998). This is, models of multi-domain physical systems (containing, e.g., electrical, mechanical, thermal, hydraulic or control subcomponents), which are mathematically described using differential and algebraic equations, and events. This type of model is known as hybrid-DAE model. The continuous-time part of the model can be described in Modelica using equations (non-causal modeling) and algorithms. In addition, Modelica includes functionalities for discrete-event management, such as *if* expressions to define changes in the structure of the model, and *when* expressions to define event conditions and the actions associated with the defined events (Otter et al., 1999).

In some cases, the application of a discrete-event modeling formalism facilitates the development and description of the discrete-event part of hybrid models. Modelica libraries have been developed for supporting different discrete-event modeling formalisms, including State Graphs (Otter et al., 2005), Petri nets (Mosterman et al., 1998) and DEVS (Sanz et al., 2010). These libraries can be used together with other Modelica libraries in order to compose multi-formalism hybrid models.

The DEVS (Discrete EVent Systems specification) formalism allows the modular and hierarchical specification of discrete-event systems (Zeigler, 1976). The Parallel DEVS formalism (Chow and Zeigler, 1994) is an extension of DEVS. DEVSLib (Sanz, 2010) is a full-fledged Modelica library that facilitates the description of discrete-event models according to the Parallel DEVS formalism and provides components to interface with continuous-time models, which can be composed using other Modelica libraries. DEVSLib can be freely downloaded from (Urquia et al., 2012).

DEVS graph (Zeigler et al., 1994) is a graphical notation for describing atomic DEVS models. Graph-based notations have the advantage of allowing the modeler to think about the problem in a more abstract way. For this reason, the use of the DEVS graph notation makes model behavior specification easier.

Two new Modelica libraries will be presented in this paper. The first library, GGADLib, supports the DEVS graph notation in Modelica. The second Modelica library, UDPLib, allows sending and receiving data using the User Datagram Protocol (UDP). GGADLib uses UDPLib. As a result, DEVS graph models composed using GGADLib can receive and send messages (input and output events, according to DEVS terminology) through the network using UDP. This feature allows Modelica DEVS graph models to communicate with hardware and with other models, e.g., with models developed using other languages and tools, and running in other computers.

The DEVS graph notation will be described in the following section. Next, the implementation and use of the UDPLib and GGADLib Modelica libraries will be discussed. Finally, these libraries are applied to assess an obstacle avoidance controller for the e-puck mobile robot (E-puck, 2012).

DEVS GRAPH NOTATION

A DEVS graph model is formally defined by the following tuple of seven elements (Wainer, 2009):

$$\text{GGAD} = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \qquad (1)$$

Proceedings 26th European Conference on Modelling and Simulation ©ECMS Klaus G. Troitzsch, Michael Möhring, Ulf Lotzmann (Editors)
ISBN: 978-0-9564944-4-3 / ISBN: 978-0-9564944-5-0 (CD)

The model may have input and output ports to communicate with other models. The tuple elements

$$X = \{(pI, x) \mid pI \in IPorts, x \in X_p\} \quad (2)$$
$$Y = \{(pO, y) \mid pO \in OPorts, y \in Y_p\} \quad (3)$$

represent the set of input ports ($IPorts$) and values (X_p), and the set of output ports ($OPorts$) and values (Y_p), respectively. The set of sequential states is represented by S,

$$S = B \times P(V) \quad (4)$$

where B represents the set of model states, also known as model phases

$$B = \{b \mid b \in Bubbles\} \quad (5)$$

and V represents the intermediate state variables of the model and their values

$$V = \{(v, n) \mid v \in Variables, n \in \mathbb{R}_0\} \quad (6)$$

The internal (δ_{int}) and external (δ_{ext}) transition functions, and the output (λ) and time-advance (ta) functions have the same meaning as in DEVS models (Zeigler et al., 2000). At any time the system is in some state $s \in S$. If no external event occurs, the system will stay in state s for time $ta(s)$. When the $ta(s)$ time expires, the system outputs the value $\lambda(s)$ and changes to state $\delta_{int}(s)$. If an external event $x \in X_p$ occurs before the expiration time, $ta(s)$, then the system changes to state $\delta_{ext}(s, e, x)$, where e is the elapsed time since the last transition.

The model graphical description is composed of bubbles, arcs and labels. Bubbles represent model phases. Each bubble includes an identifier and a state lifetime. Arcs represent transitions between states: dotted lines represent internal transitions and full lines represent external transitions. Labels placed beside external transitions indicate the corresponding transition conditions, while labels placed beside internal transitions show the associated output events.

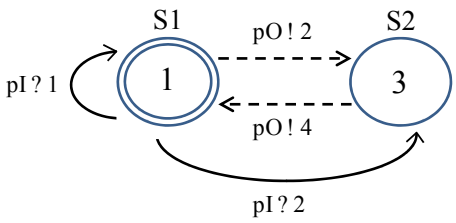

Figure 1: Atomic model defined as a DEVS graph.

The atomic model shown in Figure 1 illustrates this notation. The model has two phases: S1 and S2. The S1 bubble is composed of two concentric circumferences. This indicates that S1 is the initial phase, i.e., the model is in S1 when the simulation starts. The number written into each bubble is the corresponding phase lifetime.

The $p!v$ notation indicates that an output event of v value is generated through the p port. For instance, pO!2 means that an event with 2 value is sent through pO port.

Trigger conditions of external transitions are logical expressions. When the logical expression is evaluated to true, then the corresponding transition is executed. The $p?v$ expression is true when the v value is received in the p input port. In this model, if an event with 1 value is received in pI while in S1, an external transition to S1 is executed. If the event value is 2 while in S1, an external transition to S2 is executed.

THE UDPLib MODELICA LIBRARY

The UDPLib Modelica library is composed of two model classes, *inputUDPport* and *outputUDPport*, and two packages, *src* and *Examples* (see Figure 2a). The *src* package contains Modelica functions that are used by *inputUDPport* and *outputUDPport*. These functions are not intended to be used directly by the library users. The *Examples* package contains some use examples.

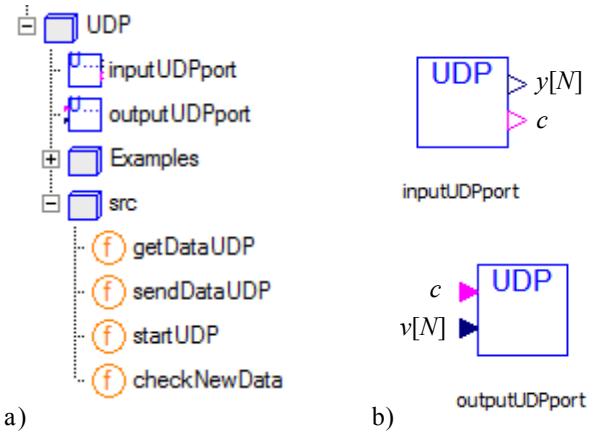

Figure 2: a) The UDPLib Modelica library; and b) *inputUDPport* and *outputUDPport* models.

The *outputUDPport* model allows to send data using the UDP protocol. The port number and IP address are model parameters, whose values can be modified when the model is instantiated. The model interface is composed of the following two connectors (see Figure 2b): an input Boolean variable, c, and an input array of N real variables, v, whose size (N) is a model parameter. When the value of c changes from false to true and vice versa, the N values of the v variable array are sent through the network to the designated IP address and port.

The *inputUDPport* model allows to receive data, using the UDP protocol, from a specified IP address and port number, which are model parameters. The model interface is composed of two connectors (see Figure 2b): an output array of N real variables, y, whose size (N) is a model parameter, and an output Boolean variable, c. The model checks for newly arrived data with a certain sampling period of simulated time, which is a model parameter (the default value is 0.01 seconds). When a new

data (an array of N real values) is received from the network, the value of the Boolean variable c is changed and the received data is assigned to the y array. Therefore, y is an array of N piecewise constant variables which contains the last received data. The changes in the value of c indicate the time instants when data have been received.

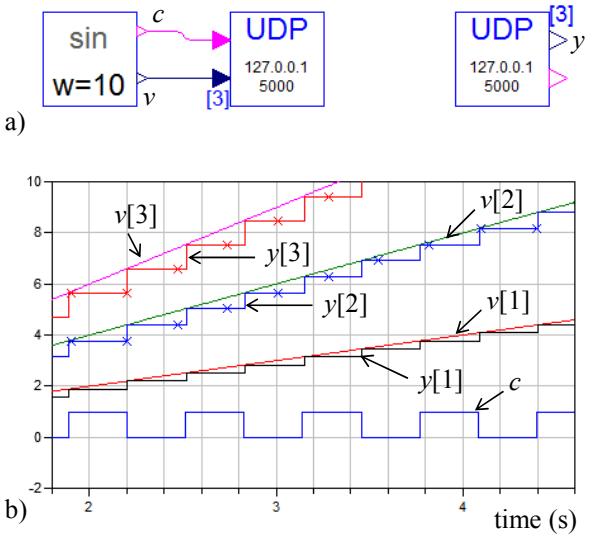

Figure 3: a) Model; and b) detail of simulation result.

The simple model shown in Figure 3a is used to illustrate the use of *inputUDPport* and *outputUDPport*. The left-most component sets the value of the c Boolean variable and the three elements ($N = 3$) of the v array as follows:

$$c = \sin(w \cdot time) > 0 \quad (7)$$

$$v = \{time, 2 \cdot time, 3 \cdot time\} \quad (8)$$

where the $time$ variable represents time and the value given to the w parameter is 10. When the value of c changes (from false to true and vice versa), the three values of the v array are sent to port number 5000, IP address 127.0.0.1.

The component of the *inputUDPport* class checks the port number 5000, IP address 127.0.0.1, for more incoming data every 0.01 seconds of simulated time. The received data are assigned to the y array. The simulated time is synchronized with real time. A detail of the simulation result is shown in Figure 3b.

In general, several instances of the *inputUDPport* and *outputUDPport* classes can be present in a model. Also, instances of these classes can be used to communicate several models running in the same and in different computers.

THE GGADLib MODELICA LIBRARY

The GGADLib library contains predefined Modelica components that facilitate to compose DEVS graphs models. These components include bubbles, to describe the model states, and internal and external transitions. The library is shown in Figure 4a. Ready-to-use components are placed at the higher hierarchical level of the library. These components inherit from the components included within the *src* package. The *Examples* package contains examples of use.

Model states are represented as bubbles in DEVS graph notation. To this end, two classes are provided in the library: *InitialState* and *State* (see Figure 4b). The *InitialState* class allows to specify the model state at the simulation start time. Therefore, a model has to contain one and only one instance of the *InitialState* class. The other model phases are described using instances of the *State* class. *InitialState* and *State* have the same superclass, *src.State*.

The *InitialState* and *State* classes have two parameters: an identifier, which stores the state name, and the state lifetime, whose default value is one second. A Boolean local variable, *active*, indicates whether the model is in the state (*active*=true) or not (*active*=false). The simulated time elapsed since the last transition is stored in the e local variable. This variable allows detecting when the state lifetime has been consumed.

InitialState and *State* have two connectors: one represented as a black filled triangle (*connectorA*) pointing toward the bubble, and another represented as a hollow triangle (*connectorB*) pointing outwards the bubble. Transitions to the state have to be connected to the first connector, while transitions from the state have to be connected to the second connector.

Components describing internal and external transitions are provided in the library. A transition takes place from an initial state to a final state. The *connectorB* of the initial state has to be connected to the *connectorB* of the transition component, and the *connectorA* of the final state has to be connected to the *connectorA* of the transition component. Common features to internal and external transitions are modeled in the *src.Transition* class, which is the superclass of the *InternalTransition* and *ExternalTransition* classes (see Figures 4c and 4d).

The *InternalTransition* class describes an internal transition and the generation of the output event associated to the transition. The internal transition is triggered when its initial state is active and the state lifetime has been consumed. The *send* and *data* connectors describe the output event generation (see Figure 4c). When the transition takes place, the value of the *send* Boolean variable changes and the *data* real variable is set to the event value. By default, the value of the output event is a component parameter (i.e., the output functions returns a single constant value). This by-default behavior can be easily modified, so that output event values can be evaluated from any Modelica function.

The *InternalTransitionUDP* class provides additional features. It sends the event value to a predefined port and IP address, using the UDP protocol. *InternalTransitionUDP* has been defined by connecting the *InternalTransition* and the *outputUDPport* classes. The icon and

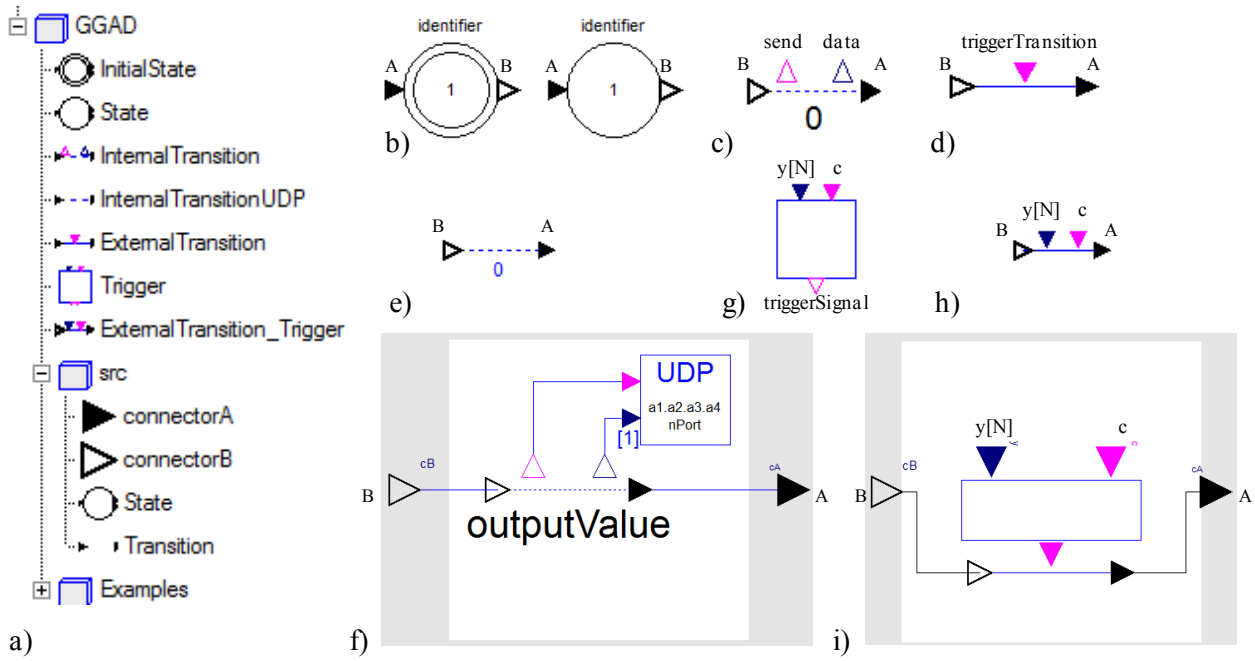

Figure 4: a) The GGADLib Modelica library; b) *InitialState* and *State* classes; c) *InternalTransition* class; d) *ExternalTransition* class; e) *InternalTransitionUDP* class; f) diagram of the *InternalTransitionUDP* class; g) *Trigger* class; h) *ExternalTransition_Trigger* class; and i) diagram of the *ExternalTransition_Trigger* class.

diagram of the *InternalTransitionUDP* class are shown in Figures 4e and 4f, respectively.

The *ExternalTransition* class facilitates describing external transitions. The transition is triggered when the Boolean value of the *triggerTransition* input connector changes (see Figure 4d) and the transition initial state is active.

Frequently, the trigger condition is the result of evaluating a logical expression. The *Trigger* partial class facilitates describing the trigger condition in these cases (see Figure 4g). The interface of this partial class is composed of an input Boolean variable (c), an array of input real variables (y) and an output Boolean variable (*triggerSignal*). When the c value changes, the value of *triggerSignal* is evaluated from an expression that depends on the y array. The expression is specified when the class is instantiated.

The *ExternalTransition_trigger* class is implemented by connecting the *Trigger* and *ExternalTransition* classes. The icon and diagram of *ExternalTransition_trigger* are shown in Figures 4h and 4i, respectively.

Several external transitions are allowed to have the same initial state. However, if two of these external transitions are triggered simultaneously, the final state is undefined. The instances of the *src.State* class check for this condition during the simulation run. When two external transitions with the same initial state are triggered simultaneously, an error message is logged out and the simulation execution aborts.

The number of internal transitions from a state can be zero or one. The *src.State* class checks that this condition is fulfilled. If it is not fulfilled, the corresponding error message is shown when the simulation starts. In case of confluence between an internal and an external transition, only the internal transition is executed.

EVALUATION OF AN OBSTACLE AVOIDANCE CONTROLLER FOR THE E-PUCK ROBOT

The simulation study discussed in this section illustrates the use of the UDPLib and GGADLib Modelica libraries. The study objective is evaluating an obstacle avoidance controller for the e-puck mobile robot (E-puck, 2012). To this end, Modelica models of the controller and the robot movement in an environment with obstacles have been developed. The communication between the real-time simulation of these models is accomplished using the UDP protocol.

Once the controller performance has been tested using simulation, the simulated robot has been substituted by the real robot. In this case, the real-time simulation of the controller is used to control the real e-puck. The communication is implemented via the UDP protocol, using ECD++ (Yu and Wainer, 2007; Moallemi and Wainer, 2010) to interface with the e-puck. The dynamic interfaces provided by ECD++ allow for integration of external entities with the real-time model which operates as a controller for the robot. Dymola 6.1 (Dynasim, 2008) is used to simulate the Modelica models.

A hybrid model of the e-puck has been developed. The model has one discrete-time state variable, $phase$, and three continuous-time state variables: the robot position, $\mathbf{r} = (x, y)$, and orientation, θ. The

$phase$ variable has the following four possible values: $\{Stop, TurnL, TurnR, Forward\}$.

The e-puck model has an input port: *motor*. Possible event values are: $\{1, 2, 3, 4, 5, 6\}$. These events, which are generated by the controller, trigger transitions between the model phases as shown in Figure 5.

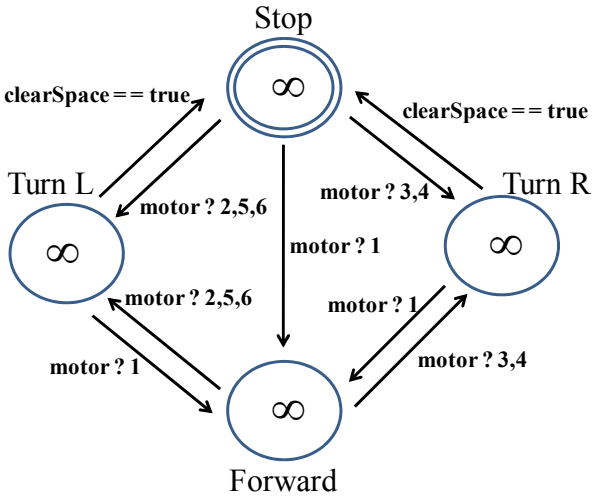

Figure 5: DEVS graph describing the phase transitions of the e-puck model.

The e-puck robot is equipped with a ring of 8 infrared proximity sensors (E-puck, 2012). The $IR[0]$, ..., $IR[7]$ readings from these sensors are the distance to obstacle measured in cm. These values are sent to the controller through an output port. In addition, the e-puck uses these readings to trigger the transition from the *TurnL* and *TurnR* phases to the *Stop* phase. The turning maneuvers stops when enough clear space is detected in front of the e-puck, i.e., when the *clearSpace* function

$$clearSpace = min(IR[0, 1, 6, 7]) > \delta \qquad (9)$$

returns true. The δ distance is a model parameter. The e-puck model calculates the sensor readings from its actual location and orientation, for a predefined environment. In this study, a $L \times H$ rectangular free space surrounded by a wall is assumed.

The e-puck position and orientation are calculated from its linear ($\mathbf{v}$) and angular (w) velocities:

$$\frac{d\mathbf{r}}{dt} = \mathbf{v}, \qquad \frac{d\theta}{dt} = w \qquad (10)$$

The velocity vector ($\mathbf{v}$) can be calculated from the velocity module (v) and the orientation (θ):

$$\mathbf{v} = |\mathbf{v}| \cdot \{\cos\theta, \sin\theta\} = v \cdot \{\cos\theta, \sin\theta\} \qquad (11)$$

The velocity module (v) and the angular velocity (w) depend on the robot phase. The velocity is zero while the e-puck is in the *Stop*, *TurnL* and *TurnR* phases. $v = V$ while *phase=Forward*. The angular velocity is $w = W$ and $w = -W$ while the e-puck is in the *TurnL* and *TurnR* phases, respectively, and $w = 0$ otherwise. The forward (V) and angular (W) velocities are model parameters.

The Modelica model of the e-puck moving in an environment with obstacles is shown in Figure 6. The *inputUDPport* component located in the upper-left corner of the figure represents the *motor* input port. It receives messages from the controller. The *sensors* component located in the lower-left corner calculates the sensor measurements from the actual position and orientation of the robot, and the obstacle position. The sensor data are sent to the controller through an output port, which is modelled using the *outputUDPport* component located in the lower-right corner of the figure. All the phase changes in this model are driven by external transitions. An arbitrary value is given to phase lifetimes: $1e10$.

The model describing the e-puck controller is shown in Figure 7. The *inputUDPport* component (upper-left corner) receives messages from the e-puck sensors. The y and c connectors of this *inputUDPport* component are connected to the y and c connectors of the *ExternalTransition_Trigger* components. For the sake of clarity, these connections are not shown in the diagram (i.e., they are drawn in white color). Components of the *InternalTransitionUDP* class are used to describe the internal transitions and the generation of output events. The output events are sent to the 5000 port of the local address.

In order to facilitate visualizing the simulation results and experimenting with the robot and controller models, an interactive graphic interface has been developed using the *Interactive* Modelica library (Martin-Villalba et al., 2012). The main window of the virtual-lab is shown in the left side of Figure 8. It contains an animated diagram of the e-puck location and orientation. The robot moves within an $L \times H$ rectangular area surrounded by walls. The length (L) and width (H) of this area can be changed interactively. Check boxes allow to show and hide plots of the robot location and orientation versus time, and the events generated by the controller and the sensors. The virtual-lab plot window displaying the time evolution of the robot position and orientation is shown in the right side of Figure 8.

CONCLUSIONS

Two new Modelica libraries have been presented: UDPLib and GGADLib. UDPLib facilitates sending and receiving arrays of real values using the UDP protocol. This feature allows communicating the Modelica models with hardware and with other models, e.g., with models developed using other languages and tools, and running in other computers. GGADLib supports the DEVS graph notation in Modelica. GGADLib uses UDPLib, facilitating the development of DEVS graph models that send and receive events using the UDP protocol. The UDPLib and GGADLib Modelica libraries can be freely downloaded from http://www.euclides.dia.uned.es/

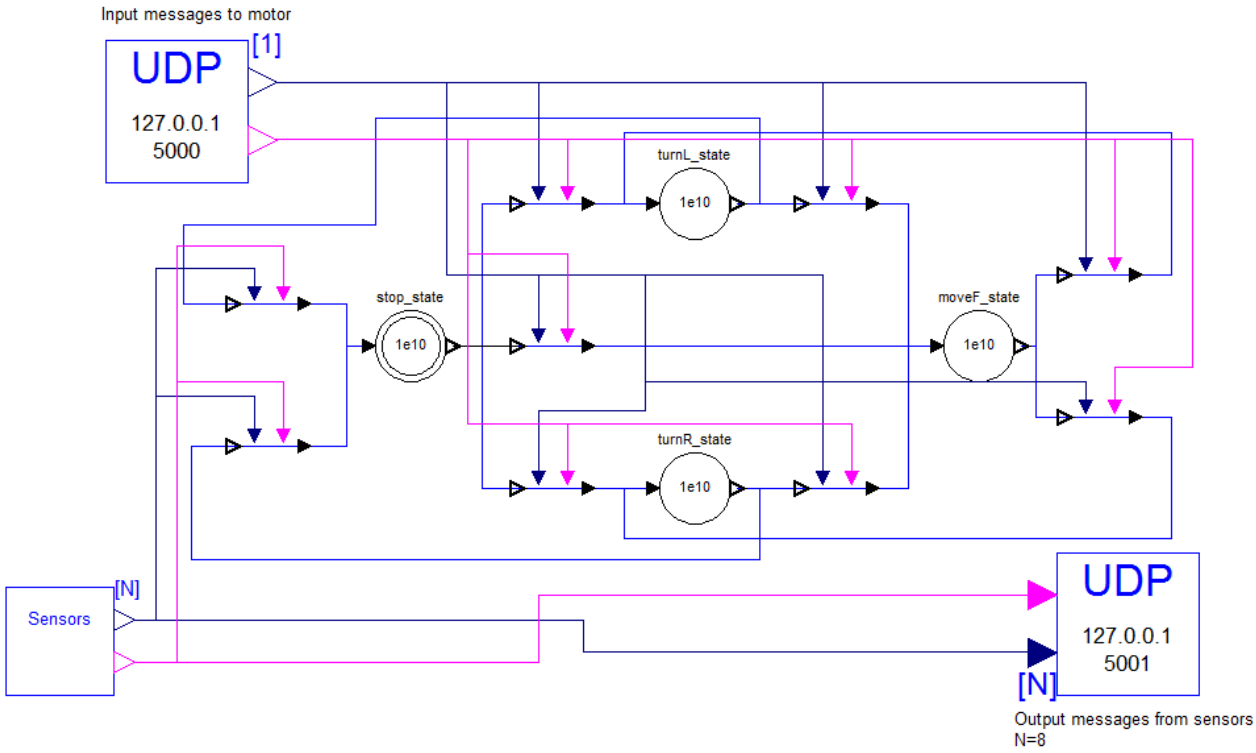

Figure 6: Model of the e-puck and its environment.

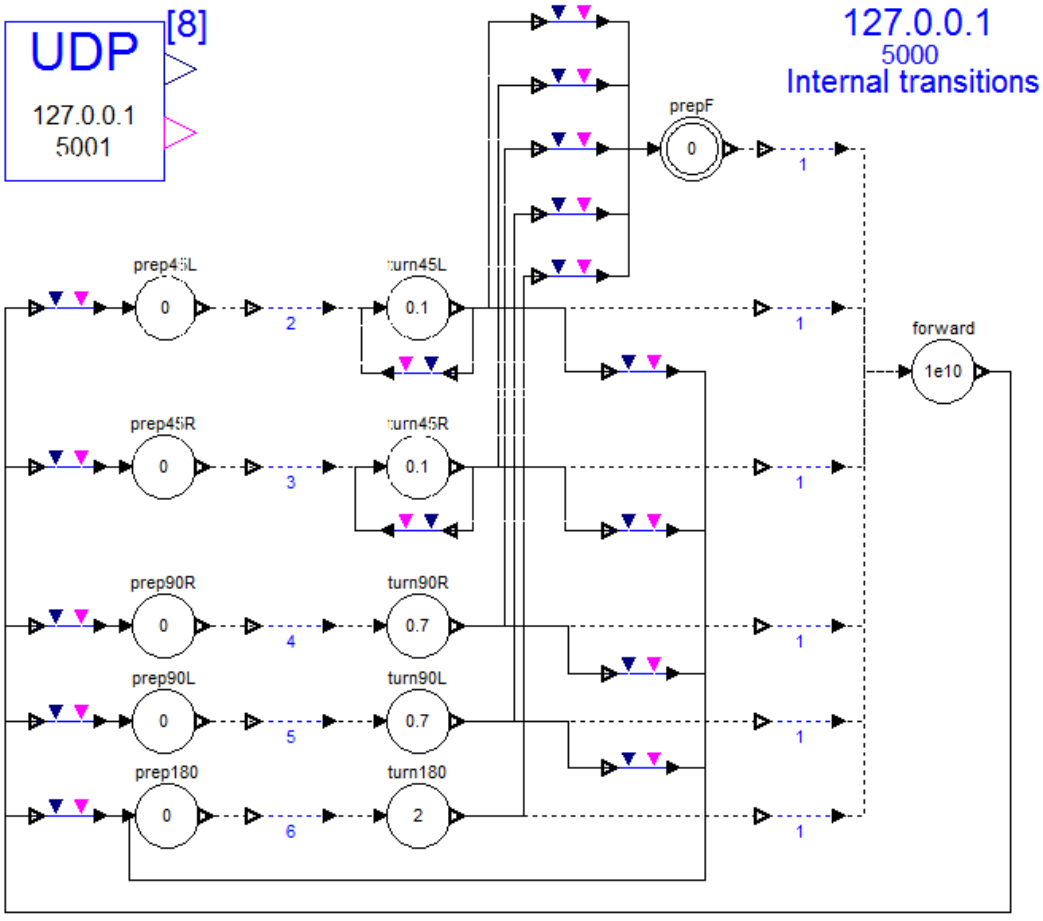

Figure 7: Model of the e-puck controller.

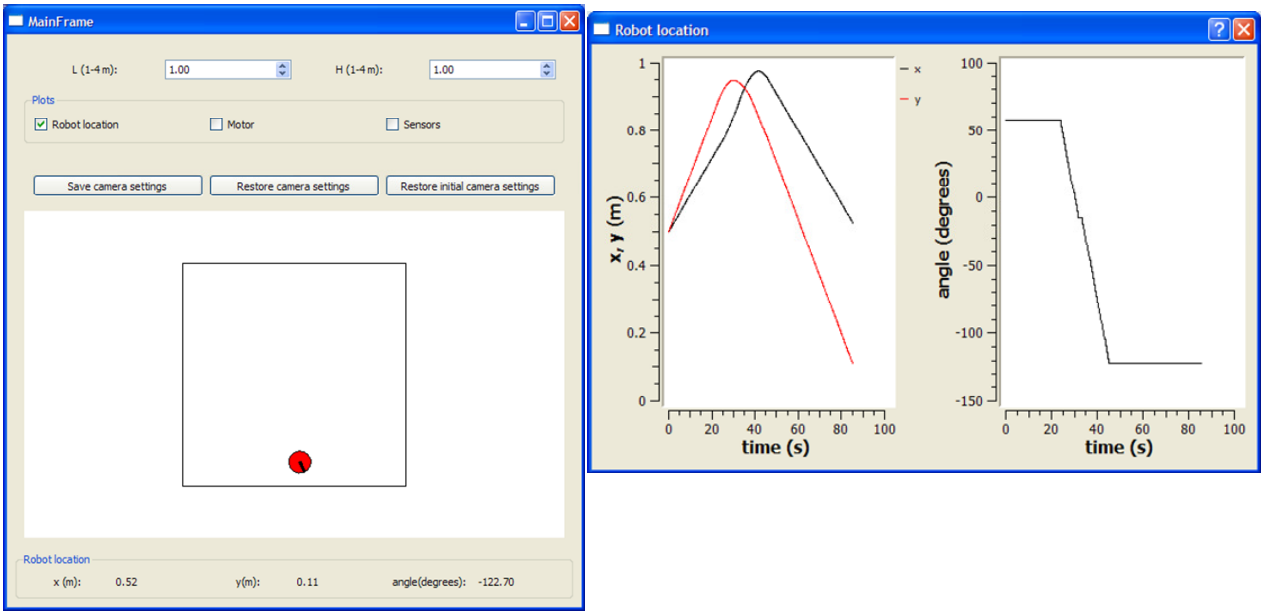

Figure 8: Main window of the virtual-lab for evaluating the e-puck controller (left). Virtual-lab plot window displaying the time evolution of the robot position and orientation (right).

ACKNOWLEDGEMENTS

This work has been supported by: Universidad Nacional de Educación a Distancia (UNED) under 2010V-PUNED/0001 grant; Programa Nacional de Movilidad de Recursos Humanos del Plan Nacional de I+D+i 2008-2011 del Ministerio de Ciencia e Innovación de España; and NSERC.

REFERENCES

Åström, K. J., Elmqvist, H., and Mattsson, S. E. (1998). Evolution of continuous-time modeling and simulation. In 12^{th} *European Simulation Multiconference*, Manchester, UK.

Chow, A. C. H. and Zeigler, B. P. (1994). Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In 26^{th} *Winter Simulation Conference*, San Diego, CA, USA.

Dynasim (2008). *Dymola - User's manual*. Dynasim AB, Lund, Sweden.

E-puck (2012). Website of EPFL education robot. http://www.e-puck.org/.

Martin-Villalba, C., Urquia, A., and Dormido, S. (2012). Development of virtual-labs for education in chemical process control using Modelica. *Computers and Chemical Engineering*, 39:170–178.

Moallemi, M. and Wainer, G. A. (2010). Designing an interface for real-time and embedded DEVS. In *2010 Spring Simulation Multiconference (SpringSim'10), DEVS Symposium*, Orlando, FL, USA.

Mosterman, P. J., Otter, M., and Elmqvist, H. (1998). Modelling Petri Nets as local constraint equations for hybrid systems using Modelica. In *Summer Computer Simulation Conference*, Reno, Nevada, USA.

Otter, M., Årzén, K.-E., and Dressler, I. (2005). StateGraph - A Modelica library for hierarchical state machines. In 4^{th} *Int'l Modelica Conference*, Hamburg-Harburg, Germany.

Otter, M., Elmqvist, H., and Mattsson, S. E. (1999). Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle. In 10^{th} *IEEE Int'l Symposium on Computer Aided Control System Design*, Hawaii, USA.

Sanz, V. (2010). *Hybrid System Modeling Using the Parallel DEVS Formalism and the Modelica Language*. PhD thesis, Dept. Informática y Automática, UNED, Madrid, Spain.

Sanz, V., Urquia, A., Cellier, F., and Dormido, S. (2010). System modeling using the parallel DEVS formalism and the modelica language. *Simulation Modelling Practice and Theory*, 18:998–1018.

Urquia, A., Martin-Villalba, C., Rubio, M., and Sanz, V. (2012). Some free modelling & simulation resources. http://www.euclides.dia.uned.es/.

Wainer, G. A. (2009). *Discrete-Event Modeling and Simulation: a Practitioner's Approach*. CRC Press, Taylor and Francis.

Yu, Y. H. and Wainer, G. A. (2007). ECD++: an engine for executing DEVS models in embedded platforms. In *2007 Summer Computer Simulation Conference (SCSC'07)*, San Diego, CA, USA.

Zeigler, B. P. (1976). *Theory of Modelling and Simulation*. John Wiley & Sons, Inc.

Zeigler, B. P., Praehofer, H., and Kim, T. G. (2000). *Theory of Modeling and Simulation*. Academic Press, Inc., Second edition.

Zeigler, B. P., Song, H. S., Kim, T. G., and Praehofer, H. (1994). DEVS framework for modelling, simulation, analysis, and design of hybrid systems. In *Hybrid Systems'94*.