Component-Oriented Interoperation of Real-Time DEVS Engines

Mohammad Moallemi, Gabriel Wainer
Dept. of Systems and Computer Engineering
Carleton University. Centre of Visualization and Simulation (V-Sim)
1125 Colonel By Dr. Ottawa, ON. Canada.
{moallemi, gwainer}@sce.carleton.ca

Federico Bergero
[1]Laboratorio de Sistemas Dinámicos
Facultad de Ciencias Exactas, Ingeniería y Agrimensura. Universidad Nacional de Rosario. CIFASIS–CONICET.
Riobamba 245 bis (2000) Rosario, Argentina
bergero@cifasis-conicet.gov.ar

Rodrigo Castro[1,2]
[2]Departamento de Computación
Universidad de Buenos Aires.
Facultad de Ciencias Exactas y Naturales.
Ciudad Universitaria, Pabellón I.
(1428) Buenos Aires, Argentina.
rcastro@dc.uba.ar

Keywords: Discrete-Event Simulation, DEVS, Embedded Systems, Real-Time Simulation and Control, Simulation-Driven Engineering

Abstract
Model reuse and interoperability are cost and effort saving solutions for the simulation-driven development of embedded real-time systems. Different embedded systems share the same components (e.g. motors, sensors, actuators, controllers, etc), and remodeling them is costly in terms of time and effort. Instead, by combining different existing models, developers can improve productivity. To do so, we here present a generic lightweight interface for message transfers between DEVS models running on different DEVS-based tools. The idea is to allow defining component-based models to be deployed on different tools collaborating in real-time. The components work autonomously as separate DEVS models, and exchange messages at the input-output level over a network infrastructure. We present a proof of concept implementation in which we interfaced ECD++ and PowerDEVS, to DEVS-based tools.

1. INTRODUCTION

Real-time application development has evolved rapidly because of the fast growing use of automated systems. These applications have usually posed interesting challenges due to the complexity of the tasks executed. As the size and complexity of the application grows, the design process tends to demand increasingly challenging multidisciplinary development efforts. Real-time systems with hard constraints require a more robust verification mechanism, due to their critical applications [1].

A solution that has been successful tackling these problems is the adoption of Modeling and Simulation (M&S) methodologies to help the development task. In particular, M&S methods using formal backgrounds have shown promising results in making these multidisciplinary systems development tasks manageable. Although M&S is often used in the early stages of real-time application development, when the focus of the project moves towards the target implementation, the early-simulated models are usually abandoned. Instead, a M&S-Driven Engineering approach [2] can deal with these activities. The idea is to use M&S at every step in the application development (including design, development, testing, and deployment). M&S is not only a foremost component in the development, but also goes further by utilizing the simulated models as the final target architecture and also offers reuse of the existing components. This leads to reduced cost and effort, while enhancing system capabilities and improving the quality of the final product. The use of formal M&S approaches provides even more advantages as theory allows model verification even before the actual model is implemented by using formal model verification techniques [3][4]. This technique also provides a reliable test-bed for applications that are impractical or impossible to verify under the actual operating conditions. Deploying the same simulation models on the target platform guarantees effective simulation-based validation of the product. Our method is based on DEVS [5] (Discrete Event System Specification) a formal modeling approach that was originally defined for discrete-event simulation. DEVS has been extended to support M&S development for real-time applications [6][7][8][9], and several DEVS-based modeling tools are available offering graphical design interfaces for easier design and verification of the models [9] [12][14][15].

In the present work, we introduce a practical, generic and lightweight interface for deploying real-time solutions communicating DEVS models implemented in different tools. We present our approach applied to sensible example models produced for the ECD++ [11] and PowerDEVS [12] toolkits

Our main motivation is to follow a Hardware-In-The-Loop approach where simulators themselves see each other as distributed real-world devices (black boxes), interacting solely at the network messaging level. The main contribution of this work is to provide a proof of concept showing that distributed M&S-based control applications can be implemented interfacing ECD++ and PowerDEVS relying on a very lightweight network-centric mechanism; reusing previously implemented models for driving mobile robots (ECD++) and reusing previously specified DEVS controllers (re-implemented in PowerDEVS).

Another motive for this work is to introduce a collaboration technique between discrete and continuous M&S-

based systems under DEVS specifications. PowerDEVS is a natural choice for implementing continuous and hybrid controllers in a methodological way in which, it deploys DEVS to approximate continuous systems and to solve differential equations by using numerical methods like the Quantized State System (QSS) [16]. Our goal is to benefit from the formal and hierarchical features of DEVS to integrated the discrete models in ECD++ with continuous and hybrid ones in PowerDEVS.

2. DEVS

The DEVS formalism permits defining hierarchical models, capable of interacting with each other and responding to the external events. The structure is composed of **atomic** and **coupled** components, in which the atomic components are the main behavioral and processing entities while the coupled components are responsible for maintaining the hierarchical structure and the couplings between the components.

A DEVS atomic component is formally defined by:

$AM = < X, S, Y, \delta_{ext}, \delta_{int}, \lambda, ta >$, where:

- X: a set of external input event types
- S: a sequential state set
- Y: an output set
- δ_{ext}: $Q \times X \rightarrow S$, an external transition function
- Where Q is the total state set of $M = \{(s, e) | s \in S$ and $0 \leq e \leq ta(s)\}$
- δ_{int}: $S \rightarrow S$, an internal transition function
- λ: $S \rightarrow Y$, an output function
- ta: $S \rightarrow R^{+}_{0,\infty}$, a time advance function which maps each state to a time interval

A coupled component connects the basic components together in order to form a new model. This component can itself be employed as a component in a larger coupled model, thereby allowing the hierarchical construction of complex models. The coupled model is defined as:

$CM = <X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select>$

, where:

- $X = \{(p, v) \mid p \in IPorts, v \in Xp\}$ is the set of input ports and values;
- $Y = \{(p, v) \mid p \in OPorts, v \in Yp\}$ is the set of output ports and values;
- D is the set of the component names

The component coupling is subject to the following requirements. *External input coupling* (EIC) connects external inputs to component inputs: $EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$, *External output coupling* (EOC) connects component outputs to external outputs: $EOC \subseteq \{((d, op_d), (N, op_N)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$, *Internal coupling* (IC) connects component outputs to component inputs: $IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$, and *SELECT*: $2^M \rightarrow M$ is a tie-breaking selector.

2.1. DEVS-Based Tools

ECD++ is a general-purpose object-oriented software able to execute DEVS models in real-time and allows users to develop hardware-in-the-loop applications being able to integrate real-world phenomena with DEVS-based environments. ECD++ deploys real-time services provided by the underlying real-time kernel to execute the DEVS abstract simulation algorithm. Its extensions allow for modeling and execution of real-time systems with tight dedaline constraints.

PowerDEVS is a software tool for DEVS modeling and simulation oriented to the simulation of hybrid systems. Among other features, it can perform real-time simulations permitting the design and implementation of synchronous and asynchronous digital controllers. Combined with its continuous system simulation library (based on the QSS quantization-based numerical methods) PowerDEVS is an efficient tool for real-time simulation of physical systems. In presence of strong discontinuities, it simulates sensibly faster than classic tools based on traditional numerical integration algorithms.

Due to these features, PowerDEVS has many potential users in the real-time automatic control community, mostly composed by non-DEVS-users. To tackle this gap, PowerDEVS offers a graphical interface very similar to those of the most popular tools for modeling, simulation and implementation of real-time controllers (Simulink, Scicos, etc.)

3. RELATED WORKS

The idea of distributed simulation using DEVS has been followed in previous works [17][16][18][19] where sophisticated middleware are proposed to deal with issues such as synchronization, fault-tolerance and message routing, among others. While these approaches are mainly targeted for simulation only, they require heavy middleware implementation and knowledge and are only designed to execute a simulation model in a distributed manner.

A split simulation consists of a source system whose components are broken into two or more groups prior to execution. These groups of components run under separate simulators that may or may not be implemented using the same simulation engine. Previous approaches (DEVS/HLA [22], DEVS/JAVA [15], Parallel CD++ [21]) implemented middleware for making virtual-time simulators collaborate based on splitting the execution among distributed processors, but composing a single simulation engine.

Another more related approach to our work is taken by [20] which uses independent distributed real-time simulation engines that accept the internal wrapping of selected components wishing to interact with other simulation tools at a high level. Wrappers hide the components and provide a means of communication with components modeled in the foreign environments. Our approach is very similar to this one, but relies completely on network messages without requiring the concept of wrappers. This generic interface-based implementation can be easily implemented for integration of different tools, requiring minimal modification to the simulation engine.

4. THE PROPOSED APPROACH

Models developed for a specific tool can be re-implemented into another tool by following their DEVS formal specification. However, this is an error prone and time-consuming approach. A more robust and scalable strategy is to keep components implemented in their original DEVS tools and make them interact over a network-centric and distributed infrastructure. This approach excludes centralized coordinators. However, in our decentralized approach, applications resulting from the collaborative activity of distributed simulators (and other real-world devices) must be designed to be as robust as required in the presence of anomalies (e.g., packet loss, corruption, and sequence inversion). The participating DEVS engines do not need to worry about clock synchronization as it is handled at the model level.

In this approach, the output ports of a DEVS model are interfaced to input ports of another DEVS model. While the simulation engines are running in real-time, different models can join this distributed network of running models and feed from the outputs of other models while contributing their own outputs to the other models in the network. The network interface for each DEVS port can be implemented in a different way (even using different network protocols), thanks to the abstract global message structure for transfer of the DEVS outputs. This is a lightweight, decoupled, physically-based method to provide a unified notion of time advance across simulators. On the other hand, this technique excludes the interchange of messages carrying absolute time references. Thus, the component-oriented aspect of DEVS allows different coupled components of a DEVS-based system to operate autonomously following a common physical notion of time advance. This plays the role of an implicit synchronization mechanism for event transfers between DEVS tools.

The approach delegates to the modeler the responsibility of being aware about the worst case scenarios expected for many real-world non-ideal behaviors. For instance, clock crystals of the hardware platforms hosting each simulator may drift, network latencies may vary considerably depending on load conditions, and also messages can be corrupted or delivered disordered. While these and other potential problems can be tackled by adding fault tolerance mechanisms into simulator engines and/or underlying communication infrastructures, there exist applications in which they can be regarded as non-critical.

4.1. Requirements

To implement the proposed communication scheme, a global message architecture is required. The message is supposed to carry DEVS outputs of one model to the input port(s) of other model(s) over a communication layer. A DEVS output set (Y) —defined in the DEVS formal specification— (see section 2) contains the (port, value) pair, in which the port is the output port and the value is the actual output value produced by the model. We need to transfer this pair over the network and inject it to another model running on another DEVS tool as an input pair. The following are the DEVS I/O message fields:

- *Port_ID:* an integer containing the destination model's input port id. Based on this field, the receiving model delivers the input to the correct input port.
- *Value:* a character array carrying the value. The value can also be a sequence of values send to a specific input port in the destination model. The format of the input value is interpreted by the input interface at the destination model

The generic message structure allows for submitting different types of data between networked models. A message interface at each DEVS port of the model provides the embedding of the message as a network packet and its extraction at the destination. Each port owns an independent interface, which can be configured to submit and receive different types and formats of the messages.

5. EXAMPLE: E-PUCK OBSTACLE AVOIDANCE

Previous experimentation with ECD++ and DEVS models for mobile robot control applications made available a repository of target-specific low-level drivers. As controller complexity grows and new requirements arise, it became convenient to split system's design tasks into specialized and collaborative teams, reusing both experience and previously developed solutions. Following a component-based approach, we plan to split a robot control model to two main components: one for the control algorithms and other for dealing with robot-specific drivers.

As a case study, DEVS was used to develop a controller for the e-puck robot [23]. E-puck is a mobile robot with sensors and motors (see Figure 1-a). It is composed of eight infrared distance proximity sensors (IR), eight LEDs mounted on the top of the robot (see Figure 1-b), and two motors.

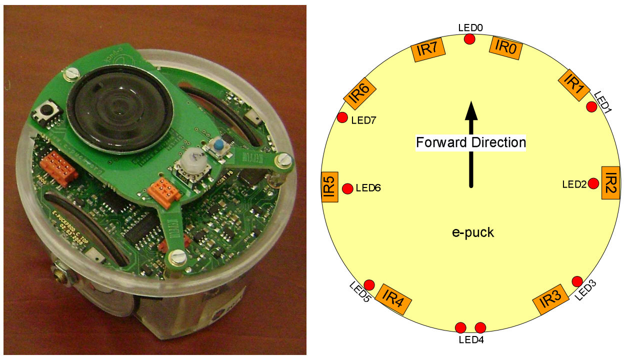


Figure 1: a) e-puck robot b) sensors and LEDs

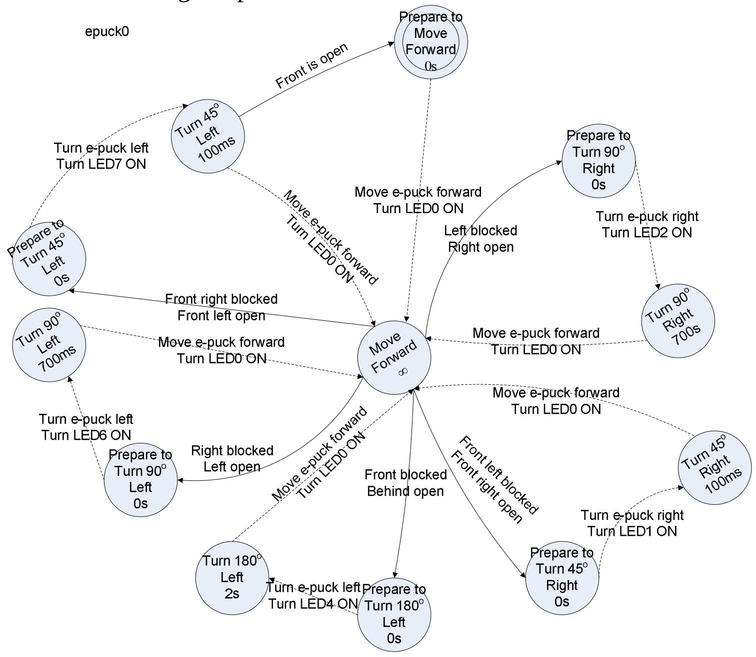
5.1. The DEVS Model

The controller model is designed to steer the robot in a field while avoiding obstacles. We have defined a DEVS model with an atomic component (e-puck0) rendering the behavior of the controller of the robot and a coupled component (Top) containing the atomic component and the couplings. There are 8 input ports (InIR0,…,InIR7) to the DEVS model, each of them receives input from one proximity sensor mounted on the robot. These input ports periodically receive the distances to the obstacles from the sensors. There are also two output ports: OutMotor: transferring the output commands to the motors and OutLED: transferring LED on/off commands to the LEDs.

Based on the inputs received from the censors, the controller takes the following different decisions: *move forward*, *turn 45 degrees left*, *turn 45 degrees right*, *turn 90 degrees left*, *turn 90 degrees right*, *turn 180*. Initially, the robot starts moving forward while receiving the periodic inputs from proximity sensors and analyzing them. As soon as an obstacle is detected, it performs one of the turning actions (to avoid the obstacle) based on the direction of the obstacle. The robot keeps turning until finds an empty space in front of it. The controller also uses LEDs to signal the action that is being performed.

Figure 2 illustrates the state diagram of the e-puck0 atomic component. The DEVSGraph state diagram [24] summarizes the behavior of a DEVS atomic component by rendering the states, transitions, inputs, outputs and state durations of the atomic component graphically. The continuous edges between the states represent external transitions, with the input port, the input value and any condition on the input. The dashed lines represent internal transitions with the associated outputs.

Initially, the robot moves forward and if no obstacle is detected from IR0, IR1, IR6 and IR7 (the four sensors scanning the front), it continues moving forward. As soon as an obstacle is detected, the value of IR6 sensor is specifically examined. If this sensor shows no obstacle, therefore the left corner of the robot is open resulting in a 45° turn towards the left side. Otherwise, it checks the IR1 value and if it shows an open space, the robot turns 45° to the right. If both IR1 and IR6 are blocked, the controller examines IR2 sensor and if it shows an open space, the robot performs a 90° turn to the left. The same story happens when IR2 is blocked and IR5 is open, resulting in a 90° turn to the right. If all of the sensors are blocked, the robot tries turning to the opposite direction (180°).



epuck0
Prepare to Move Forward 0s
Front is open
Turn 45° Left 100ms
Turn e-puck left Turn LED7 ON
Prepare to Turn 45° Left 0s
Move e-puck forward Turn LED0 ON
Move e-puck forward Turn LED0 ON
Prepare to Turn 90° Right 0s
Turn e-puck right Turn LED2 ON
Left blocked Right open
Turn 90° Right 700s
Front right blocked Front left open
Move e-puck forward Turn LED0 ON
Turn 90° Left 700ms
Move e-puck forward Turn LED0 ON
Move Forward ∞
Move e-puck forward Turn LED0 ON
Turn e-puck left Turn LED6 ON
Prepare to Turn 90° Left 0s
Right blocked Left open
Move e-puck forward Turn LED0 ON
Front blocked Behind open
Front left blocked Front right open
Turn 45° Right 100ms
Turn e-puck right Turn LED1 ON
Prepare to Turn 45° Right 0s
Turn 180° Left 2s
Turn e-puck left Turn LED4 ON
Prepare to Turn 180° Left 0s

Figure 2: E-puck atomic component state diagram

5.2. The Partitioned Model

The e-puck logical controller is divided into two parts: the *Controller* and the *Driver*. The *Controller* is the main decision making unit, where the commands to avoid obstacles are generated. The *Driver* model works as a client who forwards the inputs from robot to the *Controller* and the outputs from *Controller* to the robot. The interface to the robot is part of the *Driver* model. Figure 3 illustrates the partitioned e-puck model running on two workstations.

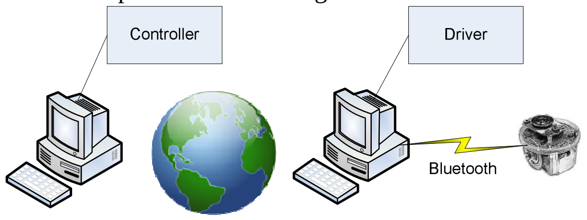


Figure 3: the partitioned e-puck model

The e-puck robot communicates with the *Driver* model running on workstation 1 via Bluetooth connection. The *Controller* model runs on another workstation communicating through network infrastructure with the *Driver*. Figure 4 depicts the e-puck collaborative DEVS model details. The e-puck *Controller* receives IR sensor values from *InIR* input port via the network and sends the motor outputs to *OutMotor* output port, which is forwarded to the *Driver* model. The *Driver* receives the IR sensor values from the e-puck robot through eight input ports, and submits them to the *Controller* model by serializing them through one output port. This method reduces the network traffic while encapsulating all the values into one network packet. The *Controller* model does not deal with LED commands, while the e-puck *Driver* model generates these commands based on the motor commands received from the *Controller*.

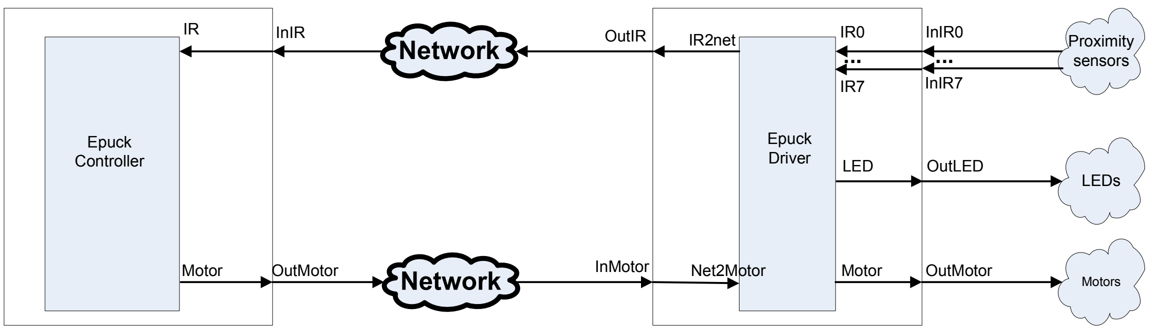


Figure 4: e-puck controller collaborative DEVS model

5.3. Implementation of The Partitioned Model

The *Controller* model is implemented on PowerDEVS and the *Driver* on ECD++. We use UDP network protocol for message transfer over an Ethernet network. We have chosen UDP over TCP for its simplicity, and since the experiments were done on a local network, the chances of loosing a UDP datagram were negligible.

5.4. The ECD++ part (Robot Drivers)

ECD++ provides generic user-implemented interfaces for DEVS model's border ports. Using this feature, the model can interact with the external world (e.g. hardware and network) by overriding an abstract C++ driver function for each port

We implemented the *Driver* model (Figure 4) in which the *OutIR* and *InMotor* DEVS ports were interfaced with the network and *InIR0*, …, *InIR7*, *OutLED*, and *OutMotor* were interfaced with the robot hardware. The *Driver* is initially in *idle* state waiting for the periodic inputs from IR sensors. As soon as it receives the first value from an IR sensor, the former buffers it until it receives the inputs of all sensors. Finally, it forwards the inputs as an array of values embedded in a network packet via the *OutIR* port to the *Controller* running on PowerDEVS. The *Driver* stays in *idle* state listening to the inputs from IR sensors and from *InMotor* port, where the motor commands are received from the *Controller*. The *Driver* generates the appropriate LED commands based on the received motor commands and forwards them to the robot. Therefore, a generic Controller model running on a different simulator with different platform is used to control a specific robot with different platform, tools, and interfaces. Each DEVS output is associated with an action on the robot. The driver functions of the robot output ports (*OutLED* and *OutMotor*) submit the commands to the robot via Bluetooth connection. An embedded program on the robot will carry out the commands on the robot hardware.

The following is a code snippet of the implementation of *InMotor* input port driver function on ECD++ (the other DEVS port driver functions are implemented in a similar method):

```
bool InMotor::pDriver(Value &value){
...
3  if  (recvfrom(s,  &buff,  BUFLEN,  0,
   (struct   sockaddr   *)   &si_other,
   &slen)==-1){
4  ...
5      return false;
6      }
7  ...
8  network_msg result;
9  memcpy((void*)&result,(void*)&buff,si
   zeof(result));
10 int i;
11 memcpy((void*)&i,(void*)result.value,
   sizeof(int));
12 value = i;
13 return true;
14 }
```

Line 1 is the header of the driver function, which is supplied by a *byrefrence* parameter ("value") which will be filled with the input received from the network. Line 3 shows the blocking UDP receive function where the **buff** parameter will be filled by the network message. The *network message* struct (defined in 4.1) declared in line 8 is filled in line 9 with the message received. The actual motor output is extracted from this message in line 11 and is assigned to "value" parameter. The input driver functions are separate real-time threads, which are only responsible for grabbing inputs from the environment (network).

5.5. The PowerDEVS part (Robot Controller)

The PowerDEVS part implements the DEVSGraph shown in Figure 2 (excluding the LED outputs). Any DEVSGraph model can be directly converted into a PowerDEVS atomic model by the following method:

- The state of the atomic is defined by an **enum** variable **s** indicating the actual DEVSGraph state (this variable will have as many values as DEVSGraph states) and **sigma**, a real value variable to hold the state duration.
- For each edge from state **s1** to state **s2** (**s2** having a duration **t2=ta(s2)**) we add a case in the internal/external function:
  ```
  if (s==s1) { s=s2; sigma=t2; }
  ```
- If the edge is from an external transition, we have to add a check to see if the input value is the one that triggers the transition. If the edge is from an internal transition, we have to emit the output event(s). In PowerDEVS, emitting multiple events in one transition is prohibited, so we have to emit them one at a time (in multiple internal transitions).

The PowerDEVS model can be seen in Figure 5 where the IR block receives the value of the 8 IR sensors from the e-puck and forwards them to the *Controller*. The *Controller* depending on values, change its state and emits to the e-puck through the *OutMotor* block one of the following commands: MOVE_FORWARD, TURN_45_LEFT, TURN_90_LEFT, TURN_45_RIGHT, TURN_90_RIGHT, or TURN_180.

We introduced two new blocks for connecting the PowerDEVS side with ECD++. These blocks are *NetSend* and *NetReceive*. In Figure 5, the IR block is an instance of *NetReceive* and it receives the data from ECD++ and injects them to the PowerDEVS simulation. The *OutMotor* is an instance of *NetSend*, it receives the events from the *Controller* and sends them to ECD++ through the network.

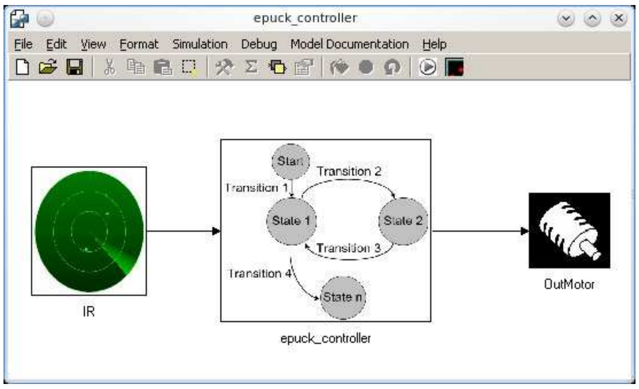


Figure 5: PowerDEVS controller model

The *NetSend* block is responsible for sending the events from PowerDEVS to ECD++. It has a parameter to indicate to which UDP port should send the message (the IP address is fixed in the code).

The *NetReceive* block is in charge of receiving the UDP messages from ECD++ and forward them to the correct PowerDEVS atomic component.

The PowerDEVS simulation engine will receive the UDP message, read the port_id field from the payload and forward it to the corresponding *NetReceive* block. To this end, the simulation engine internally holds a mapping from DEVS ports to *NetReceive* blocks, created at initialization.

5.6. The common network messages semantics

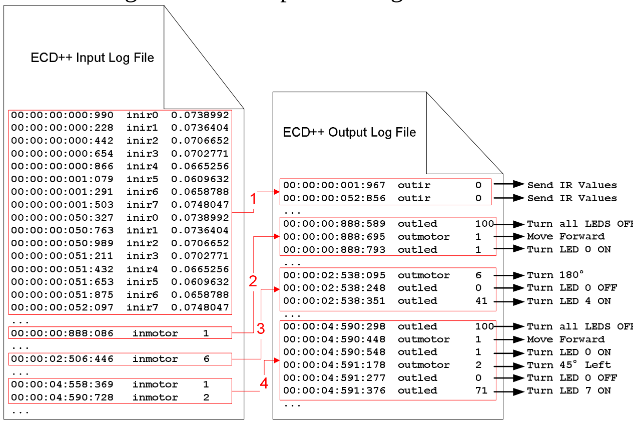
As said before, the content of the UDP message is a fixed-size buffer were sender and receiver have to agree on a format. The *Value* field of the global message (section 4.1) is used with the following semantics for different ports:

- For messages going from ECD++ to PowerDEVS the buffer contains 8 doubles (8 bytes each - IEEE 754) representing the magnitude of each IR of the e-puck.
- For messages going from PowerDEVS to ECD++, we only send one 4 byte integer indicating which command to send to the motors.

6. RESULTS

We conducted various experiments implementing the example model presented in section 5. To show the results of the two simulators collaborating over a network, we present a log file of the experiment with real-time timestamps, and discuss the results.

Figure 6 shows the input and output log files of ECD++ simulator. The input log file records all the real-time incoming data (from the environment) to the model's input ports while the output file saves all the outputs of a DEVS model (with microseconds precision). The inputs and associated outputs are marked with red boxes in the figure. In the first box of the input file, two series of the IR sensor values inputted at time zero and after 50 milliseconds are shown (the IR sensor inputs are received every 50 milliseconds.) The first box of the output file shows the output to the *OutIR* port, which triggers the output driver associated to this port to send the array of inputs containing the values of the eight IR sensors. Therefore, when all of the IR values are received, they are forwarded to the *Controller*. Box 2 of the input file shows an input signal received from *InMotor* port containing value "1", which is interpreted in box 2 of the output file with the accompanying LED commands (added by the *Driver*). The same sequence happens in box 3 where the robot has found an obstacle and the associated IR sensor values are forwarded to the *Controller*, hence the *Controller* is instructing the robot to spin 180 degrees.


ECD++ Input Log File

```
00:00:00:000:990  inir0  0.0738992
00:00:00:000:228  inir1  0.0736404
00:00:00:000:442  inir2  0.0706652
00:00:00:000:654  inir3  0.0702771
00:00:00:000:866  inir4  0.0665256
00:00:00:001:079  inir5  0.0609632
00:00:00:001:291  inir6  0.0658788
00:00:00:001:503  inir7  0.0748047
00:00:00:050:327  inir0  0.0738992
00:00:00:050:763  inir1  0.0736404
00:00:00:050:989  inir2  0.0706652
00:00:00:051:211  inir3  0.0702771
00:00:00:051:432  inir4  0.0665256
00:00:00:051:653  inir5  0.0609632
00:00:00:051:875  inir6  0.0658788
00:00:00:052:097  inir7  0.0748047
...
00:00:00:888:086   inmotor    1
...
00:00:02:506:446   inmotor    6
...
00:00:04:558:369   inmotor    1
00:00:04:590:728   inmotor    2
...
```

ECD++ Output Log File

```
00:00:00:001:967  outir       0    Send IR Values
00:00:00:052:856  outir       0    Send IR Values
...
00:00:00:888:589  outled      100  Turn all LEDS OFF
00:00:00:888:695  outmotor    1    Move Forward
00:00:00:888:793  outled      1    Turn LED 0 ON
...
00:00:02:538:095  outmotor    6    Turn 180°
00:00:02:538:248  outled      0    Turn LED 0 OFF
00:00:02:538:351  outled      41   Turn LED 4 ON
...
00:00:04:590:298  outled      100  Turn all LEDS OFF
00:00:04:590:448  outmotor    1    Move Forward
00:00:04:590:548  outled      1    Turn LED 0 ON
00:00:04:591:178  outmotor    2    Turn 45° Left
00:00:04:591:277  outled      0    Turn LED 0 OFF
00:00:04:591:376  outled      71   Turn LED 7 ON
...
```

Figure 6: ECD++ input and output log files

A selected progression of UDP messages interchange taken from the simulation results can be seen in the sequence diagram of Figure 7. The messages going from the Controller to the Drivel model are motor commands (the same as in the log in Figure 6) while the messages from the Drivel to the Controller are the values from the 8 IR sensors. According to these values, the *Controller* is in charge of deciding whether the front is blocked. (see message labeled *"Front is Blocked"* in Figure 7).

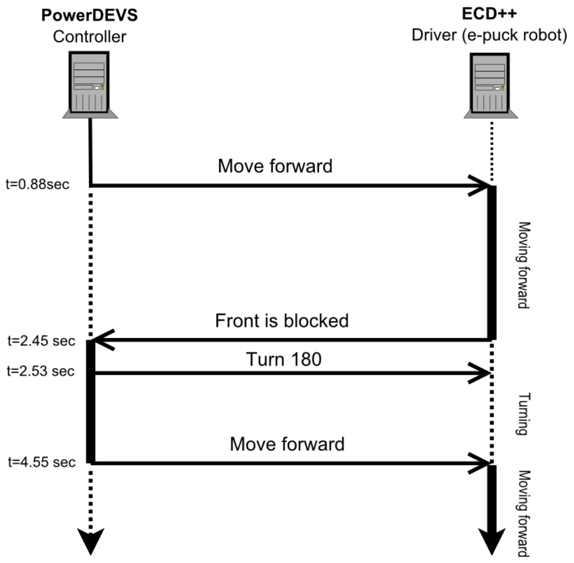


Figure 7: Sequence diagram of the simulation

A video of the collaborative e-puck model in action can be viewed online in [25].

7. CONCLUSION

We introduced a generic lightweight interface for network I/O message transfers between DEVS models running on different DEVS-based tools. We showed the suitability of our approach by reproducing experiments with an e-puck robot and a collision avoidance application, monitoring correctness for behavior (qualitatively) and network massaging (quantitatively). The robot succeeded to perform obstacle detection and direction changing when the original DEVS-based system was split into two distributed real-time models Controller and Drivers running on PowerDEVS and ECD++, respectively.

Thanks to the unambiguous formal specification of DEVS, the problem of splitting a model into components deployable to distributed real-time tools can be confined into an implementation layer, preserving original model specifications. Also thanks to DEVS formal definition, the task of migrating subcomponents previously developed for ECD++ to PowerDEVS can be synthesized into a repeatable procedure, minimizing effort and errors. Another potential advantage of interfacing ECD++ to PowerDEVS is the collaborative execution of discrete and continuous systems under DEVS specifications.

Nevertheless, in the approach presented in this work some limitations must be observed. Messages across simulators cannot bear time references into their semantics, leaving the synchronization to the modeling level, where the models should be in proper states while transferring data. Yet, for those applications where synchronization is mandatory, additional logical layers for exchanging timing signals can be implemented on top of the networking framework presented in this work. This represents the basis of our next research steps and efforts.

References

[1] Liu, J. "Real-Time Systems". Prentice-Hall, 2000.

[2] G. Wainer, E. Glinsky, P. MacSween "A Model-Driven Technique for Development of Embedded Systems Based on the DEVS Formalism". In Model-driven Software Development - Volume II of Research and Practice in Software Engineering. S. Beydeda and V. Gruhn Eds. Springer-Verlag. 2005.

[3] S. Merz, N. Navet. "Modeling and Verification of Real-Time Systems: Formalisms and Software Tools". John Wiley & Sons, publishing Inc. 2008.

[4] H. Saadawi, G. Wainer. "Verification of real-time DEVS models". Proceedings of DEVS Symposium 2009. San Diego, CA. 2009.

[5] B. Zeigler, T. Kim, H. Praehofer. "Theory of Modeling and Simulation". Academic Press 2000, ISBN-10: 0127784551.

[6] Hong J. S, Song H. H, Kim T. G. and Park K. H "A Real-Time Discrete Event System Specification Formalism for Seamless Real-Time Software Development" 1997, Springer Netherlands.

[7] Hu, X.; Zeigler, B.P. "Model Continuity in the Design of Dynamic Distributed Real-Time Systems", IEEE Transactions on Systems, Man And Cybernetics— Part A: Systems And Humans, 35: 6, pp. 867- 878, November, 2005.

[8] Cho S. M. and Kim T. G. "Real-Time DEVS Simulation: Concurrent, Time-Selective Execution of Combined RT-DEVS Model and Interactive Environment" In Proceeding of 1998 Summer Simulation Conference, Reno, Nevada.

[9] Moallemi, M..; Wainer, G. "Designing an Interface for Real-Time and Embedded DEVS", Proceedings of Symposium of Theory of Modeling and Simulation, Orlando, FL, 2010.

[10] Wainer, G. "CD++: a toolkit to define discrete-event models". Software, Practice and Experience. Wiley. Vol. 32, No.3. pp. 1261-1306. November 2002.

[11] YU, J.; WAINER, G. "ECD++: a tool for modeling embedded applications". In Proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007.

[12] Federico Bergero and Ernesto Kofman. "Powerdevs: A Tool for Hybrid System Modeling and Real-time Simulation". SIMULATION, 2010.

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

[14] Traoré, M. 2008, "SimStudio: a next generation modeling and simulation framework". Proceedings of SIMUTools 2008. Marseille, France.

[15] Sarjoughian, H; Zeigler, B. 1998, "DEVSJAVA: Basis for a DEVS-based collaborative M&S environment" proceedings of the International Conference on Web-based Modeling & Simulation, San Diego, CA.

[16] Francois Cellier and Ernesto Kofman "Continuous System Simulation" Springer, New York, 2006.

[17] Cho, Y. K.; Hu, X.; Zeigler, B.P. "The RTDEVS/CORBA Environment for Simulation-Based Design of Distributed Real-Time Systems", Simulation: Transactions of the Society for Modeling and Simulation International, Volume 79, Number 4, 2003.

[18] Kim, Y.J. and Kim, T.G. "A heterogeneous distributed simulation framework based on DEVS formalism", Proceedings of the Sixth Annual Conference On Artificial Intelligence, Simulation and Planning in High Autonomy Systems, pp 116-121, 1996.

[19] Kim, Yong Jae and Kim, Jae Hyun and Kim, Tag Gon "Heterogeneous Simulation Framework Using DEVS BUS" SIMULATION, 2003

[20] Lombardi, S., G. Wainer, and B. P. Zeigler. "Interoperation of DEVS models in DEVS/C# and CD++" Proceedings of SISO Fall Interoperability Workshop, Huntsville, AL, 2006

[21] Troccoli, A.; Wainer, G. "Implementing Parallel CD++". Proceedings of the Annual Simulation Symposium. Orlando, FL. 2003.

[22] H. Sarjoughian, B. Zeigler "DEVS and HLA: Complimentary Paradigms For M&S?", Transactions of the SCS Vol. 17, pp. 187-197, 2000.

[23] E-puck robot website available at: http://www.e-puck.org/.

[24] G. Christen, A. Dobniewski and G. Wainer, "Modeling State-Based DEVS Models in CD++". In Proceedings of MGA, Advanced Simulation Technologies Conference 2004 (ASTC'04). Arlington, VA. U.S.A.

[25] Shared e-puck model video, available at: http://www.youtube.com/arslab#p/u/12/iRqrwkPL-kQ, accessed Jan. 2010.