Int. J. Engng Ed. Vol. 26, No. 4, pp. 938–949, 2010
Printed in Great Britain.

0949-149X/91 $3.00+0.00

Teaching Embedded Control Systems*

GONZALO FARIAS
Department of Computer Science and Automatic, National University for Distance Education (UNED), 28040 Madrid, Spain. E-mail: gfarias@bec.uned.es

KARL-ERIK ÅRZÉN
Department of Automatic Control, Lund University, SE-221 00 Lund, Sweden. E-mail: karlerik@control.lth.se

ANTON CERVIN
Department of Automatic Control, Lund University, SE-221 00 Lund, Sweden. E-mail: anton@control.lth.se

SEBASTIÁN DORMIDO
Department of Computer Science and Automatic, National University for Distance Education (UNED), 28040 Madrid, Spain. E-mail: sdormido@dia.uned.es

FRANCISCO ESQUEMBRE
Department of Mathematics, Murcia University, 30071 Murcia, Spain. E-mail: fem@um.es

This paper introduces a novel approach to building virtual laboratories of embedded control systems using TrueTime and Easy Java Simulations. TrueTime is a freeware MATLAB/Simulink based tool commonly used to simulate embedded control systems. Easy Java Simulations is a popular authoring tool that facilitates the creation of pedagogical simulations. According to the proposed approach, authors use TrueTime to develop the simulation of an embedded control system and then move to Easy Java Simulations to link the system to a sophisticated graphical user interface that provides the visualization and user interaction of the virtual lab required for pedagogical purposes. The combination of these two tools conforms a powerful, yet simple, approach to the creation of effective pedagogic simulation of real-time control systems.

Keywords: control education; real-time; embedded control systems; virtual labs; TrueTime; EJS

1. INTRODUCTION

REAL-TIME SYSTEMS AND CONTROL THEORY both have long, but separated, traditions. Since the beginning of the 1970s, there has been extensive research on real-time scheduling. However, very little of this work has focused on control tasks. Also, digital control theory, with its origins in the 1950s, has not addressed the problem of shared and limited resources in the computing system until very recently [1, 2, 3]. Instead, it is commonly assumed that the controller executes as a simple loop in a dedicated computer [4]. This misunderstanding has frequently led to wrong assumptions, such as that the computation delay of the controller is fixed or that the controller deadline is always critical. On the contrary, many control algorithms have a varying computation time (e.g. model predictive controllers), and a single missed deadline does not necessarily cause system failure.

A new interdisciplinary approach is currently emerging where control and real-time issues are discussed at both design levels. The development of algorithms for this co-design of control and real-time systems requires new tools. One of these new tools is TrueTime, a freeware MATLAB/Simulink based simulator for networked and embedded control systems that has been developed at Lund University since 1999. However, simulations of Simulink models typically lack interactivity and visualization, two crucial features from a pedagogical point of view [5, 6]. Without these features, simulations can be hard-to-understand learning objects.

Indeed, in engineering, a typical analysis of the system response comes from the features of its output signals (waveform, periodicity, etc.), and this analysis is usually neither direct nor intuitive because output signals are actually not human-readable. Therefore, instead of using only signal plots, teachers should add a richer level of graphical content to their pedagogical simulations in order to produce more intuitive and natural learning objects. A second important issue is the fact that much of the analysis is usually done in an *off-line* way. That is, signals are obtained and observed only when the simulation has finished, with students rarely interacting with the system, changing parameters or inputs, while the simula-

* Accepted 15 February 2010.

tion runs. On the other hand, an *on-line* (or *on-the-fly*) interaction provides users with the possibility of modifying some inputs or parameters of the system in run-time, which allows students to understand input/output relationships more quickly and also to appreciate the degree of influence that any input has in the global system response [7].

Nowadays, fortunately, information and communication technologies let us apply all these interesting features to the field of control education. Some interesting applications on engineering education can be found in [8–10]. However adding visualization and interactivity to an existing simulation can be a technical challenge for teachers who are not experts in computer programming: we introduce in this context the Easy Java Simulations (EJS) authoring tool. EJS is an open source software program that allows authors who do not have advanced programming skills to produce virtual laboratories (labs for short) with a high level of interactivity and visualization.

This paper focuses on providing a new approach for teachers who want to use advanced simulations of embedded control systems as part of their courses on control engineering. This approach proposes the creation of virtual labs (pedagogical simulations) with a high level of interaction and visualization using both the TrueTime and EJS software tools. Here the authors use TrueTime to develop the simulation of an embedded control system and then move to EJS to build the Graphical User Interface (GUI), which provides the required visualization and user interaction. These interactive virtual labs can significantly reduce the learning curve of the study of embedded control systems.

The paper is organized as follows. Section 2 briefly discusses core concepts of embedded control systems and introduces the TrueTime and EJS tools. Section 3 describes how these tools can be combined to create virtual labs of advanced embedded control systems. Section 4 presents two such virtual labs in detail. Section 5 discusses the pedagogical evaluation of the created virtual labs. Conclusions and further works are finally discussed in Section 6.

2. BACKGROUND

This section briefly introduces the main concepts of real-time systems. The TrueTime and Easy Java Simulations software tools are also presented.

2.1 Embedded control systems

An embedded control system consists of a computer whose specific task is to apply a control algorithm to keep a signal of a process within prescribed safety margins, despite disturbances. The control task typically executes periodically and under limited implementation resources (CPU time, communication bandwidth, energy, etc.). A system is said to be real-time if the total correctness of the operation depends not only upon its logical correctness, but also upon the time in which it is performed [11]. Real-time systems can be classified into two subcategories: *hard* real-time systems, in which the completion of an operation after its deadline may lead to a critical failure of the complete system, and *soft* real-time systems, which tolerate such lateness and may respond with decreased service quality (such as a slower reaction time or a longer settling time).

The stabilization of an inverted pendulum by moving its base back and forth (the academic version of how the Segway Personal Transporter keeps its verticality) is a simple example of a real-time system. Suppose our operation requirements specify that the pendulum must recover its verticality as soon as possible after suffering any moderate perturbation. If the sampling period of the vertical angle of the pendulum is 80 ms, with a time delay of 20 ms for the engines to act on the base, a reasonable design could require that the control algorithm is executed every 80 ms and has a worst case execution time of 60 ms. To prevent the pendulum from falling, the control algorithm must be both correctly designed and applied in time.

Real-time tasks such as our control of verticality can be periodic, aperiodic or sporadic, and are characterized by different parameters, among which are:

- *the deadline*, which indicates the maximum execution time allowed to ensure correct execution. It is common to take the period of a periodic task as its deadline;
- *the release time*, which indicates the next time instant to execute a task;
- *the finish time*, which signals when a task has finished its execution;
- *the execution time*, which is the duration of the execution of a task;
- *the period*, which indicates the amount of time after which a periodic task has to be released. For periodic tasks, the release time is always a multiple of the period;
- *the priority*, which defines the preference of execution of a task with respect to other tasks.

Typically, a control task executes in parallel with several other tasks, including other control tasks. This coexistence gives relevance to the *scheduling policy* of the system, which is the algorithm that decides which task executes at a given time. The presence of a scheduling policy introduces the *priority* parameter of a task: its preference with respect to the other tasks in the system.

In our example, the control of the pendulum's verticality would typically be a top-priority, periodic task, with a period of 80 ms and an execution time smaller than 60 ms, which makes a deadline of 80 ms reasonable. Under a scheduling policy, tasks may be in one of the three following states: *running*, *pre-empted* or blocked, and *sleeping*.

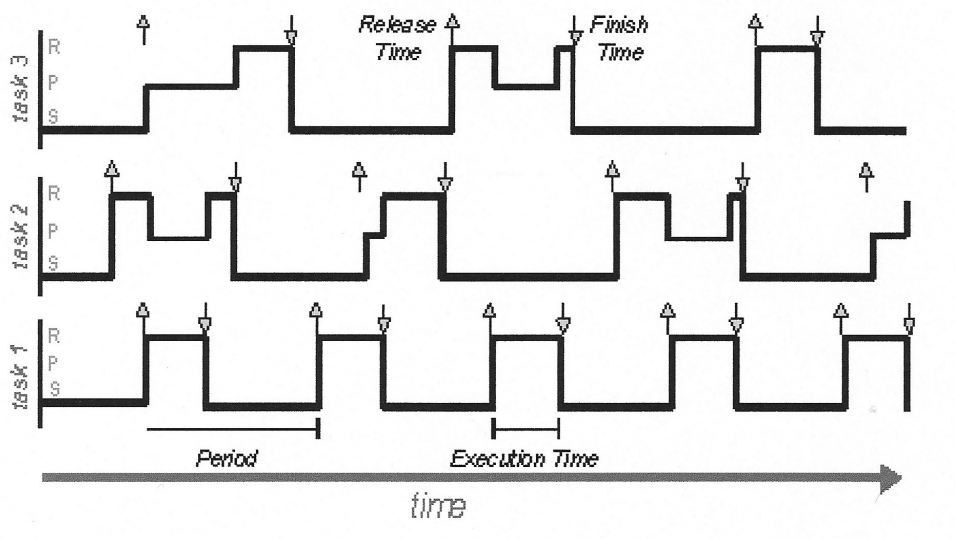

Fig. 1. Scheduling plot. Three periodic tasks are running on the same CPU: tasks 1 and 3 having the highest and lowest priority, respectively. Up arrows in a task plot indicate the release times of that task, down arrows indicate the task finish times. The initials R, P and S indicate the possible states of the tasks. Note that task 1 in this plot is never pre-empted.

Running means that the task is actually executing. Pre-empted means that the task needs to be executed, but it is not being executed because another task is running (usually one with higher priority). Sleeping indicates that the task has finished and is waiting for its next release time. A scheduling plot, such as the one shown in Fig. 1, is a graphical representation that is commonly used to illustrate the evolution of the states of the tasks in time.

The scheduling policy can be either static or dynamic. For instance, Rate Monotonic (RM) is a popular static scheduling policy that assigns the priorities of the tasks on the basis of their period: the shorter the period, the higher the priority of the task. Earliest Deadline First (EDF) is a dynamic scheduling policy that places tasks in a priority queue. Whenever a *scheduling event* occurs (e.g. when a task is released) the queue is searched and the process closest to its deadline is scheduled for execution.

2.2 TrueTime

TrueTime [12, 13] is a MATLAB/Simulink [14, 15] based simulator for networked and embedded control systems that has been developed at Lund University since 1999. The simulator software consists of a library of Simulink blocks and a collection of MEX files. The TrueTime Kernel block simulates a computer node that contains a real-time kernel that executes user-defined tasks and interrupt handlers. The TrueTime Network block is capable of simulating a variety of wired and wireless networks. TrueTime can be downloaded from *http://www.control.lth.se/truetime*.

To create a simulation using TrueTime, the plant dynamics are first modelled using ordinary Simulink blocks. Then, kernel and network blocks, which represent the computer implementation of the controller, are added to the model. For each kernel block, a set of MATLAB functions (M-files) have to be written: one function to initialize the kernel (and possible network interfaces) and one MATLAB function for each task and interrupt handler in the real-time system. To model the execution time of a task or interrupt handler, a special code function format is used.

A code function is divided into code segments as shown in Fig. 2. The execution of user code is done non-pre-emptively at the start of each segment. The code function returns the simulated execution time of the segment. Inside code functions, users can access the kernel block inputs and outputs using kernel primitives (such as *ttAnalogIn* and *ttAnalogOut*).

2.3 Easy Java simulations

EJS is a free software tool for fast creation of Java simulations with a high level of graphical capabilities and an increased degree of interactivity [16, 17]. EJS is different from most other authoring tools in that EJS is not designed to make life easier for professional programmers, but it has been conceived for science students and teachers. That is, for people who are more interested in the content of the simulation, the simulated phenomenon itself, and much less in the technical aspects

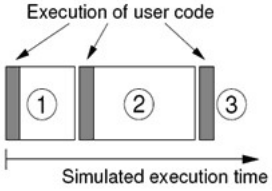

Fig. 2. TrueTime code model. The execution of task code (or user code) is modelled by a sequence of code segments that are executed in sequence by the kernel.

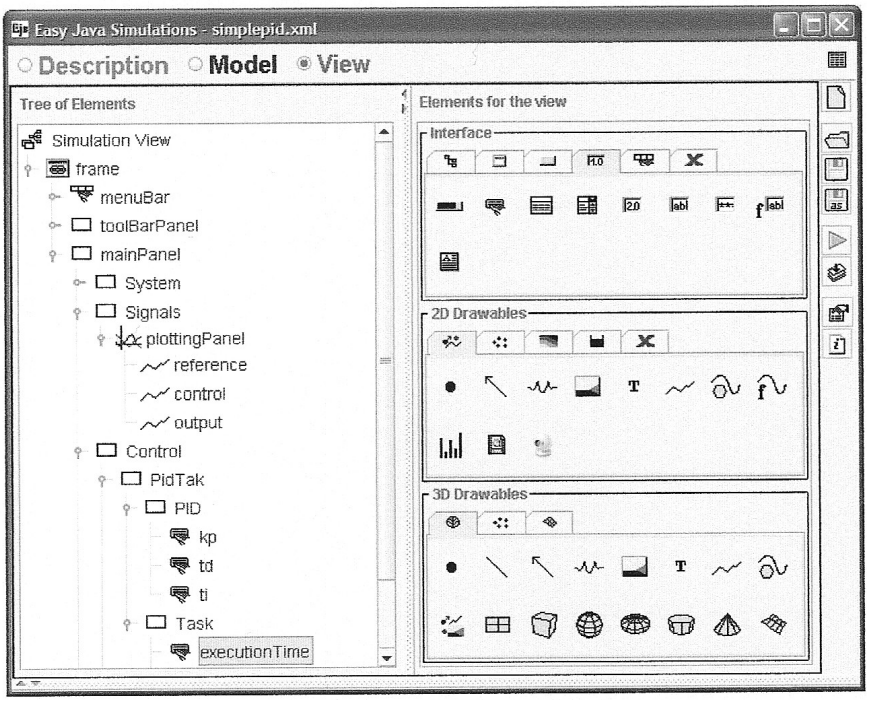

Fig. 3. Graphical user interface of Easy Java Simulations.

needed to build the simulation. EJS can be downloaded from *http://www.um.es/fem/Ejs*. EJS structures a simulation into two main parts: the *Model* and the *View*. The *Model* can be described by means of pages of Java code, ordinary differential equations, or by interfacing to other programs (external applications) such as MATLAB/Simulink [18], Scilab [19] and Sysquake [20]. The *View* provides the visualization of the simulated system and the user interface elements required for user interaction. The view of the simulation is constructed using elements from a set of predefined components to build the tree-like structure of Fig. 3. Model and View are then easily interconnected so that any change in the state of the model is automatically reflected by the view, and vice versa, in order to provide a dynamic and interactive visualization of the system.

3. COMBINING TRUETIME AND EJS

The approach proposed in this paper for the implementation of pedagogically effective simulations of real-time control systems divides the process into two phases. Authors first use TrueTime to design the embedded control system, defining the scheme of the system, the plant to be controlled, the tasks code (e.g. the controller), the tasks features (e.g. periods and priorities), the scheduling policy, the network (if needed), etc. Authors then move to EJS to build the Graphical User Interface (GUI) of the simulation, providing the required visualization and user interaction through a suitable combination of the visual elements offered by EJS. Creating simulations using either TrueTime or EJS separately is widely described in detail in the literature [2, 3, 12, 13, 16, 17]. This section describes the combined use of both tools to create virtual laboratories. The next section analyses a complete example to show the main steps that authors must complete to create an interactive simulation using our approach.

3.1 Using MATLAB/Simulink as external application for EJS

As mentioned above, EJS offers a direct link to MATLAB/Simulink. For the case of Simulink models, the link consists in allowing the connection of EJS variables to signals (such as inputs, outputs or parameters) of the blocks of the Simulink model. To set these connections between variables and signals, EJS provides a special column labelled *Connected to* in its panels of *Variables*. EJS also provides the _external Java object, which has a set of built-in methods that allow authors to read and write variables from and to the MATLAB workspace, and to execute MATLAB commands. These methods are based on the functions defined in the MATLAB Engine Library [14], which interacts with MATLAB at a low level. The three most important built-in functions are:

- **_external.eval(String cmd)**: Executes a command in MATLAB as given by the **cmd** parameter (similarly to the MATLAB *eval* function)
- **_external.getDouble(String name)**: Retrieves the value of a *double* variable from the MATLAB workspace
- **_external.setValue(String name, Object value)**: Sets the value of a variable in the MATLAB workspace.

The **_external** object also has methods to advance one or more integration steps of a Simulink model (*step(int times)*), to get the value of an array (*getArray(String name)*) or a matrix (*getArray2D(String name)*) from the MATLAB workspace, etc.

A more detailed description of the connection between EJS and MATLAB is presented in [18] and [21]. Reference [22] shows a complete example of a simulation created using this approach.

The procedure to connect EJS and MATLAB/Simulink can be summarized in the following four steps.

1. Set MATLAB/Simulink as an external application for EJS.
2. Connect EJS variables with MATLAB/Simulink variables.
3. Control the execution and access to MATLAB/Simulink variables.
4. Define the desired visualization and interactivity.

3.2 Integration of TrueTime and EJS

To build simulations that use the TrueTime–EJS combination, authors just need to follow the procedure described above for EJS–MATLAB/Simulink connections. The first two steps, setting the external application and connecting the variables, are relatively simple and are completed with a few mouse clicks. The control of the execution of MATLAB/Simulink is implemented using the EJS built-in functions described above. This step frequently consists in advancing the simulation time of the Simulink model, evaluating some MATLAB commands, and reading or writing the MATLAB variables of interest. Since there are normally many interesting variables declared for local use in the M-files of a TrueTime simulation, accessing the MATLAB variables from EJS requires re-declaring them as *global* variables. Probably, the last step of the connection procedure, defining the required visualization and interactivity, demands most of the design time. For this reason, authors should thoroughly evaluate all the requirements of the virtual lab from a pedagogical point of view before concentrating on the technical aspects of the connection process.

The described connection procedure guarantees that True-Time simulations (Simulink models and M-files) can be linked to EJS in a very direct way. However, to improve the visualization and performance of the virtual labs, authors need to consider two particular aspects of TrueTime simulations: *zero crossings evaluations* and *scheduling data*.

The first aspect is important because TrueTime models use scheduling algorithms that involve a great deal of zero crossing evaluations, which can slow down simulations considerably. To improve performance, authors need to indicate EJS that the link to the TrueTime models should update the connected variables only at fixed time intervals. This configuration will make the simulation run at a faster and also smoother way. Otherwise, there could be too much exchange of information between EJS and TrueTime, causing undesirable delays in the simulation. This update time is specified in the first step of the connection procedure.

The second aspect is related to the scheduling data. The information of the scheduling data is crucial for a successful analysis of a real-time system, because it indicates the states (running, sleeping or waiting) of a task. For this reason, authors must instruct EJS to get all samples from the scheduler to make sure that these signals are shown correctly in the virtual lab. This instruction is necessary because, by default, EJS tries to get MATLAB variables only a finite number of times. Authors must then use the *external.setWaitForEver(true)* built-in method to force EJS to wait for those particular MATLAB variables (the scheduling data) as much as needed, until they are finally available for reading from the MATLAB workspace.

4. EXAMPLES OF VIRTUAL LABS

Two virtual labs using the TrueTime-EJS integration are shown in this section. The first one simulates a periodic task that controls a simple system, and its purpose is to exemplify how to use the combination of both tools to create a virtual lab. The second virtual lab is a complete example that shows the potential of our approach to obtain sophisticated interactive simulations of embedded control system for pedagogical purposes.

4.1 Simple PID servo controller

This virtual lab uses one of the sample simulations distributed with TrueTime [13]. This example simulates a periodic PID-controller [23] embedded in a computer that controls a DC-servo process (a second-order system). The controller is the single task executing on the computer. This task is divided into two code segments: one segment to compute the control algorithm and another to send out the control action.

The TrueTime simulation uses a Simulink model that represents the complete system, and some M-files to initiate the system and to describe the code function to be executed as control algorithm. Four different implementation modes of the task are provided by this example: Built-in Task, Simulink Block, Sleep Until, and Trigger Task.

From a pedagogical point of view, the virtual lab can be used mainly to show how control performance can be affected by the computing time of the control algorithm [2]. Besides this key concept, the virtual lab also allows students to specify the implementation mode of the task, to modify the PID parameters, to change the reference, to view the output and control signals, and to control the period and the computing time of the control algorithm.

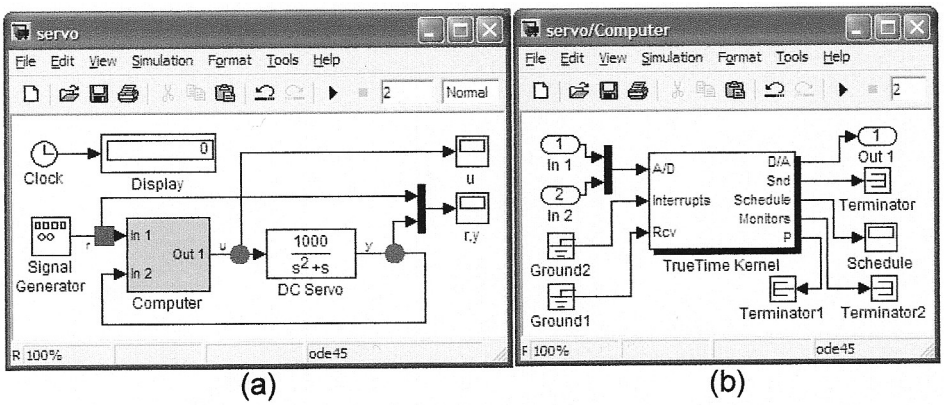

Fig. 4. An example of TrueTime simulation: (a) Simulink model; (b) blocks of the Computer submodel.

The Simulink model is shown in Fig. 4(a). Note that the feedback control is done by the *Computer* submodel, and that the DC-servo process is described by a *Transfer Fcn* block. The *Computer* submodel, shown in Fig. 4(b), uses the *TrueTime kernel* block to simulate a computer. The parameters of the *TrueTime kernel* block (see Fig. 5(a)) are used to indicate the *initialization function* (an M-file) that initiates the configuration of the computer, and also to provide an argument that represents, in this example, the implementation mode of the task. The parameters of the block *schedule* (see Fig. 5(b)) have been slightly modified from the original one, by adding the *ScopeData* variable. This modification will save the scheduling data of the task to the MATLAB workspace after every integration step.

To link the TrueTime simulation with EJS, the first step consists of selecting the Simulink model and connecting the signals to suitable EJS variables. There is a total of five EJS variables connected. The *time* and *mode* variables are connected to Simulink parameters to get the simulation time and to set the configuration mode. The *reference* variable is connected to the first input of the *Computer* block to set the reference or set point value from EJS. Finally, the *control* and *output* variables are connected to the outputs of the *Computer* and *DC servo* blocks to read the control and servo-output signals, respectively.

The selected input and outputs signals are displayed as squares and circles, respectively, in Fig. 4(a). The result of the connection process is shown in Fig. 6.

The text '*<matlab(0.01)> servo.mdl*' states that the Simulink model *servo.mdl* will be used as an external application, and that the fixed time interval for updating data (in order to improve the performance of the simulation) is *0.01*.

Once the connection step has finished, and in order to control and access MATLAB/Simulink, some modifications of the M-files are needed. The main requirement is to adapt the code functions for accessing MATLAB variables from EJS. Since the M-files of TrueTime simulations are mainly functions, the simplest way to access the local variables is to redeclare them as global variables.

The first M-file thus modified is the *initialization function,* shown in Listing 1. This M-file is used to initialize the computer where the controller (the task) is executed. The script is divided into two parts: the *initialization* code and the *switch* code.

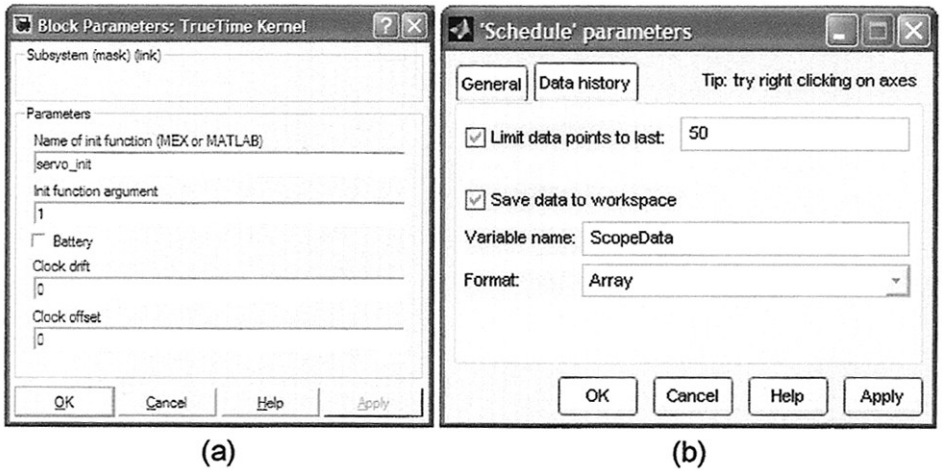

Fig. 5. Parameters of the TrueTime kernel block (a) and parameters of the Schedule block (b).

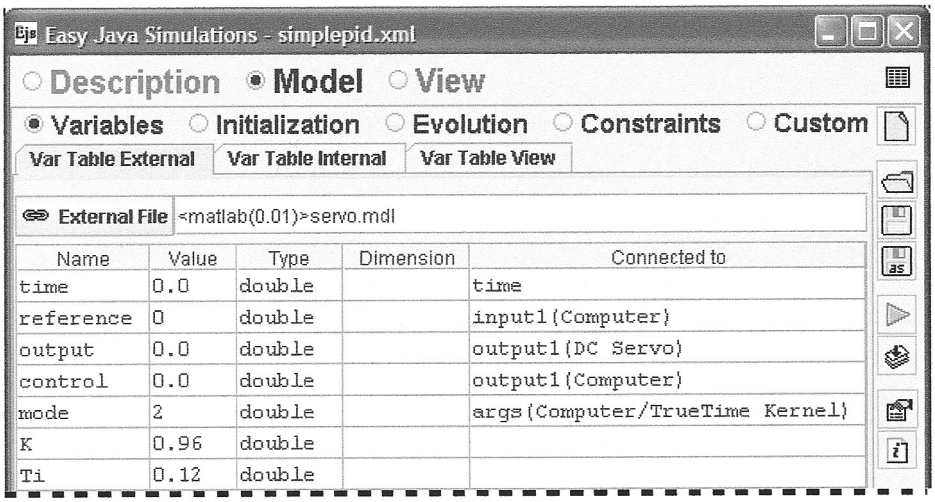

Fig. 6. Setting a link between EJS and a TrueTime model; note that the fixed time interval updating is used and that the EJS variables are connected to inputs and outputs of the Simulink blocks.

Listing 1. initialization function

```
1  function servo_init ( mode )
2  % Initialize TrueTimekernel
3
4  % nbrOfInputs, nbrOfOutputs, fixedpriority
5  ttInitKernel (2, 1, 'prioFP');
6
7  % Link To EJS
8  global period;
9  global data;
10
11 % Taskattributes
12 deadline = period;
13 offset = 0.0;
14 prio = 1;
15
16 %Create task data (local memory)
17 data.K = 0.96;
18 data.Ti = 0.12;
19 data.Td = 0.049;
20 . . .
21
22 switch mode ,
23 case 1 , % IMPLEMENTATION 1
24 % using the built -in support for periodic tasks
25 t tCreatePeriodicTask ( 'pidtask ', offset,
   period,
26 p r i o , ' pidcode1 ' , d a t a ) ;
27 case 2 , % IMPLEMENTATION 2
28 . . .
29
30 end
```

The *initialization* code comprises lines 1 to 21. This code uses *ttInitKernel(2, 1, 'prioFP')* to configure a computer with two inputs (reference and DC-servo output), one output (control signal), and a fixed priority policy. The initialization also defines and initializes some variables. Observe that the *period* and *data* variables are declared as global in lines 8–9 in order to be able to access them from EJS. The *period* variable indicates the period of the task, and the *data* variable is used by the task as a local memory to save parameters such as the gain, the integral time, and the derivative time of the PID controller.

A *switch* construction is used in lines 22–30 to execute one of the four different modes to implement the periodic task. The selected implementation depends on the *mode* argument of the *initialization function*. The first implementation mode is shown in lines 23–26, where the TrueTime *ttCreatePeriodicTask* function is used to create the periodic task. Note that the *pidcode1* M-file is the code function to be executed by the computer in this mode. Every implementation mode has a code function associated to it, but we focus only in *pidcode1* because the modifications required are quite similar for the other files.

The *pidcode1* M-file is displayed in Listing 2. The script has two parts: the *declaration* section and the *code segment* section. The first section, lines 1–4, has been modified to redefine the output of the function and to redeclare the *data* and *exectime* variables as global. The *exectimeAux* auxiliary variable is used to return to TrueTime the simulated computing time of a code segment. The second section, lines 6–19, describes the two code segments of the task.

The first code segment (lines 8–12) is used to compute the control algorithm and the second segment sends out the control action. Note that the *pidcalc* M-file is called (line 10) to compute the PID control algorithm [23], and that one of the arguments of this function is the *data* global variable. Note also that only the execution time of the first code segment can be modified from EJS (by using the *exectime* global variable) because in the second code segment (lines 15–17) the *exectimeAux* variable has a fixed value. The negative value of the execution time means that the code segment is the last segment of the task and its computing time is zero. This modified listing shows that students can modify parameters such

as the gain, the derivative and integral time, and the execution time of the first code segment from the EJS generated interface.

Listing 2. Code function modified to set a link between EJS and TrueTime. Lines with comments correspond to the original lines of the function.

```
1  function [exectimeAux,data] =
   pidcode1(seg,data)
2  % function [exectime,data] =
   pidcode1(seg,data)
3
4  global data exectime; %EJS
5
6  switch seg,
7  case 1,
8  r = ttAnalogIn(data.rChan); % Readreference
9  y = ttAnalogIn(data.yChan); %
   Readprocessoutput
10 data = pidcalc(data,r,y); % CalculatePIDaction
11 % exectime=0.002;
12 exectimeAux = exectime;
13
14 case 2,
15 ttAnalogOut(data.uChan,data.u); %
   ControlSignal
16 % exectime = –1;
17 exectimeAux = –1;
18
19 end
```

Once the M-files have been properly modified, authors move to EJS to control and access the MATLAB/Simulink model. As mentioned above, EJS simulations have two main parts: the *Model* and the *View*. Authors use the *Model* part to describe the behaviour of the system, and use the visual elements provided by the *View* part of EJS to build the GUI of the virtual lab. There are five sections in the *Model* that help authors to systematize the system description process: *Variables*, *Initialization*, *Evolution*, *Constraints,* and *Custom* (see Fig. 6). Here we explain only the first three sections (the other two are empty in our example).

The *Variables* section is used to declare the EJS variables. In our case, the main variables of the virtual lab were defined when we established the link between EJS and the TrueTime simulation (Fig. 6).

The *Initialization* section is normally used to prepare the simulation before it runs. For instance, in our virtual lab we used the code of Listing 3 to initiate TrueTime, to execute some MATLAB commands, and also to set the initial values of some variables. Note that the *_external.setWaitForEver(true)* function is used in this section to make sure that all the variables will be read from the MATLAB workspace after every simulation step.

Listing 3. Initialization code in EJS.

```
1  //Initiate TrueTime
2  _external.eval
   ('addpath([getenv('TTKERNEL')])');
3  _external.eval ('inittruetime;');
4
5  //Wait to recover Matlab variables
6  _external.setWaitForEver (true);
7
8  //Declare Global Variables
9  _external.eval ('globalperiod');
10 _external.eval ('globaldata');
11 _external.eval ('globalexectime');
12
13 //Setinitialvalues
14 _external.setValue ('exectime', 0.002);
15 _external.setValue ('period', 0.012)
16 _external.eval ('data.K='+0.96);
17 _external.eval ('data.Ti='+0.12);
18 . . .
```

The *Evolution* section contains the actions executed by EJS in every simulation step. Listing 4 shows the code used for the *Evolution* of our example. Two important actions are required in this virtual lab: stepping the Simulink model and retrieving the scheduling data.

The first action (line 3) is done invoking the *_external.step(int n)* built-in method. This method sends the values of the connected EJS variables to MATLAB, steps the Simulink model as many times as the argument indicates, and finally retrieves the values of all connected EJS variables from MATLAB. The second action (lines 6–9) reads the values of some particular variables, such as *ScopeData*, which is updated by the *Schedule* block (Fig. 5(b)) after each simulation step. To get this information, the *_external.getDoubleArray()* built-in method is used. Note that this information is retrieved separately in two EJS variables (two arrays), *scopeT* and *scopeS*. These two variables are then used by the EJS *Polygon* visual element.

Listing 4. Evolution code in EJS.

```
1  . . .
2  // Stepping the Simulink model
3  _external.step (1);
4
5  // Getting the Scheduler Data
6  _external.eval ('scheT=ScopeData(end
   –49:end,1)'');
7  _external.eval
   ('scheS=ScopeData(end–49:end,2)'');
8  scopeT = _external.getDoubleArray ('scheT');
9  scopeS = _external.getDoubleArray ('scheS');
10 . . .
```

The final step of the TrueTime-EJS combination approach concerns visualization and interactivity. To create the graphical user interface of this virtual lab, we used the visual elements provided by the *View* of EJS (Fig. 3) to build the user interface shown in Fig. 7.

To add interactivity to the virtual lab, we needed

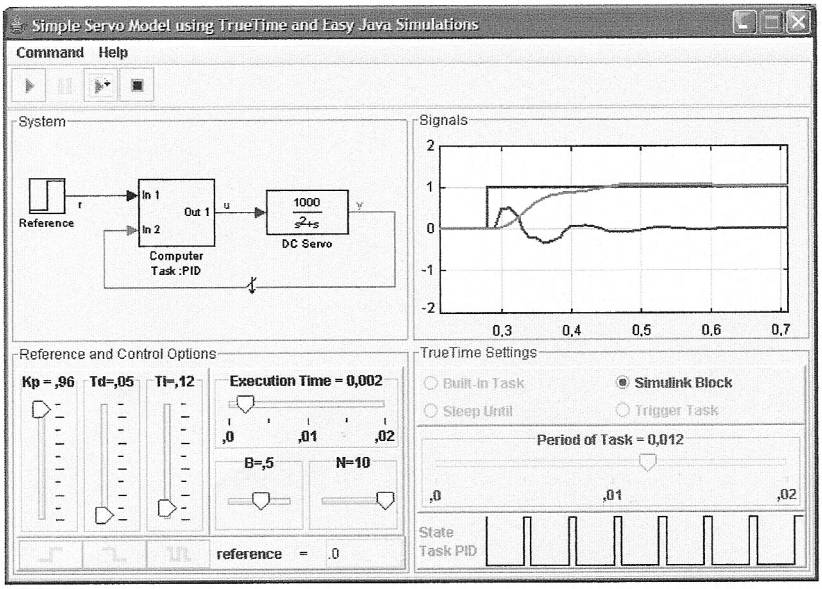

Fig. 7. Graphical User Interface of our first example. Note that it is possible to use the sliders provided to change the control parameters on-the-fly.

to define what happens when end users interact with the visual elements. For instance, note that we added sliders to allow manipulation of the PID parameters (*kp*, *ti* and *td*) and of the *execution time* of the controller. Sliders allow end users to change the values of variables in a very natural way while the simulation runs.

Figure 8 shows the procedure required to add interactivity to our GUI. The figure displays the property inspector of the *slider* that controls the execution time of the task.

We add interaction to this visual element by inputting the code to invoke when the slider is *pressed*, *dragged* or *released*. For instance, in this virtual lab, if the slider that controls the execution time is moved and then released by the user, then the value of the MATLAB *exectime* variable will be updated to the new value using the following sentence: *_external.setValue('exectime',exectime)*. Other sliders of the interface also invoke this action.

The virtual lab thus created allows end users to modify a great number of parameters of the system, such as the reference type, the control settings, the execution time of the controller, and the implementation mode of the tasks, among others. As an example of this interaction, Fig. 9 shows the performance of the controller for execution times of 5 ms and 9 ms. In both cases, the period is 12 ms and the control parameters are the same. However, the control performance of the first case is better than that of the second one.

4.2 Distributed servo control

The example of this subsection simulates the distributed control of a DCservo [13]. The example contains four computer nodes, each represented by a TrueTime kernel block, connected by a network. A time-driven sensor node samples the process periodically and sends these samples over the network to the controller node. The control task in this node calculates the control signal and sends the result to the actuator node, where it is subsequently actuated. The simulation also involves an interfering node sending disturbing traffic over the

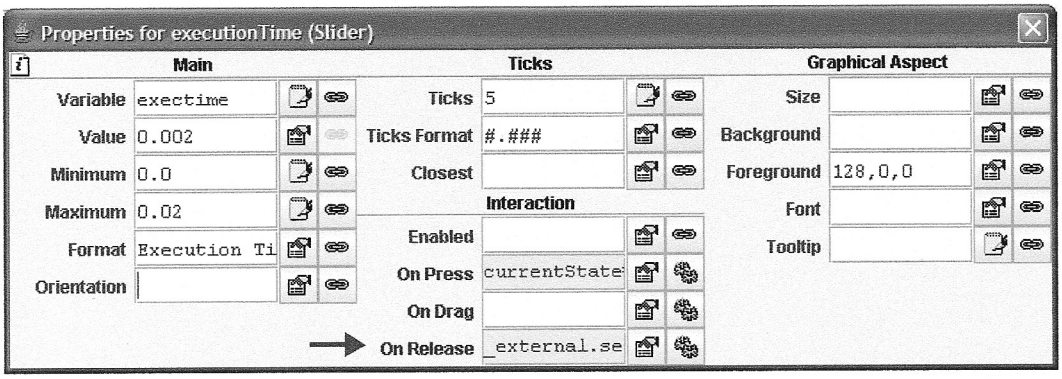

Fig. 8. Parameters of the slider that control the execution time of the task.

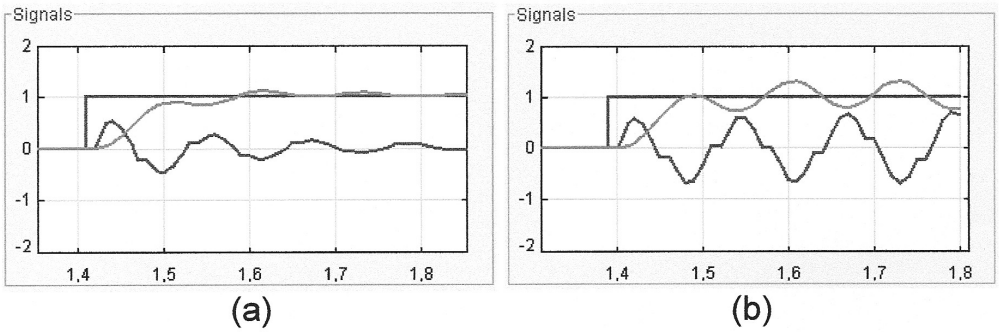

Fig. 9. Reference, control and output signals when the execution time is (a) 5 ms and (b) 9 ms.

network, and a disturbing high-priority task executing on the controller node.

This system is a bit more complex than the previous example. The main difference is that, here, we have many more M-files and also a bigger Simulink model to modify. However, the connection process required to create a virtual lab is quite similar to the previous example and, although a bit longer, the integration between both tools is still easy.

The virtual lab created is shown in Fig. 10. The main view (Fig. 10(a)) allows end users to modify the parameters of the network and the nodes. There are also three auxiliary dialogues that display a histogram of the end-to-end latency (Fig. 10(b)), show the scheduling data of the four tasks (Fig. 10(c)), and allow the user to modify the controller code (Fig. 10(d)). This last possibility gives the virtual lab great flexibility because users can test different control strategies to face the effects of the disturbances due, for instance, to the network. Note also that the controller code is written in MATLAB code, which means that end users can use any MATLAB toolbox available on their computers.

In the main view, end users can modify control parameters and also add a dummy disturbing high-priority task with different execution times. In the *Sensor* section of the same window, users can modify the measurement time and the package size. This last parameter is important in order to see the effect of the size of the package on the performance of the control. The *Interference* section allows end users to increase the bandwidth used by the interference node, which adds some disturbances to the network. The network parameters, such as the transmission rate or the loss package rate, can be modified in the *TrueTime Settings* section of the main window.

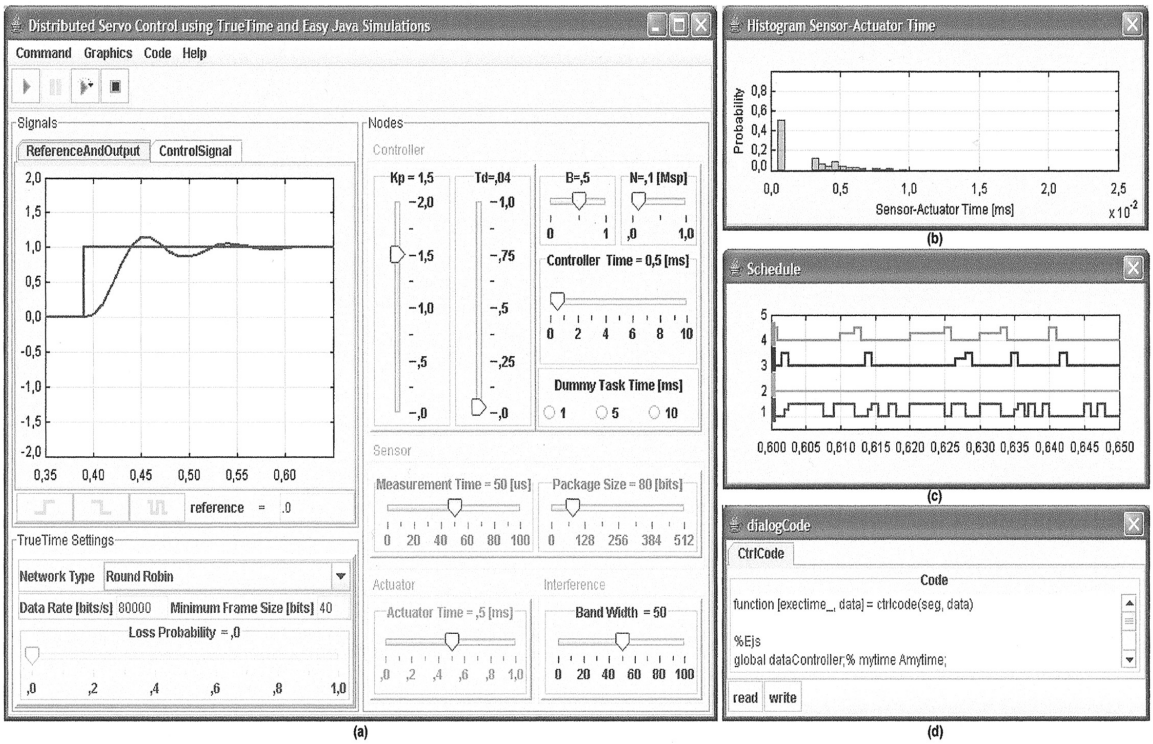

Fig. 10. Graphical user interface of the second example. Figuress show: (a) the main view, (b) histograms of sensor–actuator time, (c) the scheduler data for the four nodes, (d) the dialog window where users can read and modify the code of controller.

5. EVALUATION

The approach presented in this paper has been used to build several virtual labs of embedded control systems. Three of these labs, the two described in this article and an extended version of the first example, are in a course on Automatic Control at the National University for Distance Education (UNED). In the course, the students have to send in a report according to the required activities defined in the user's guide of the virtual lab. The students have reported that the simulations have enabled them to learn the theory at their own pace.

The course evaluation has been very satisfactory, and the students have acquired a new qualitative and practical view of their theoretical knowledge about real-time control systems. The EJS models are freely available at *http://lab.dia.uned.es/ejstruetime.*

6. CONCLUSIONS AND FUTURE WORK

Information and communications technologies open new possibilities for control education. Simulations with a high degree of interactivity and visualization provide a more natural way of learning and teaching engineering. This article introduces a new approach that combines TrueTime and Easy Java Simulations (EJS) to build this kind of virtual labs for embedded control systems.

TrueTime is a MATLAB based tool that has particular functionalities that are well suited for the simulation of embedded control systems. However, from an author's point of view, creating simulations with rich visual content using only MATLAB features demands a lot of work. We then make use of EJS because it facilities the creation of interactive simulations in Java to non-programming instructors. The combination of both tools offers teachers a powerful, yet easy to use, platform to build virtual labs of embedded control systems with a high level of interactivity and visualization.

Future work will involve the development of other virtual labs, new performance improvements, and the creation of a link between EJS and the Scilab version of TrueTime. Simulations discussed here can be found on-line at *http://lab.dia.uned.es/ejstruetime.*

Acknowledgements—The authors wish to thank Professor J. Sánchez Moreno (UNED) for his constructive and pertinent comments.

REFERENCES

1. A. Cervin, Integrated control and real-time scheduling, Ph.D. thesis, Lund Institute of Technology, 2003.
2. A. Cervin, D. Henriksson, B. Lincoln, J. Eker and K. Årzén, How does control timing affect performance?, *IEEE Control Systems Magazine*, **23**(3), 2003, pp. 16–30.
3. M. Andersson, D. Henriksson, A. Cervin and K. Årzén, Simulation of wireless networked control systems, *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC*, Seville (Spain), 12–15 December 2005, pp. 476– 481.
4. L. Dong-Jin, A laboratory course in real-time software for the control of dynamic systems, *IEEE Transactions on Education*, **49**(3), 2006, pp. 346–354.
5. B.S. Heck (ed.), Special report: Future directions in control education, *IEEE Control Systems Magazine*, **19**(5), 1999, pp. 35–58.
6. S. Dormido, Control learning: Present and future, *IFAC Annual Control Reviews*, **28**(1), 2004, pp. 115–136.
7. S. Dormido, S. Dormido-Canto, R. Dormido-Canto,J. Sánchez and N. Duro, The role of interactivity in control learning, *International Journal of Engineering Education: Especial issue on Control Engineering Education*, **21**(6), 2005, pp. 1122–1133.
8. A. Leva, F. Donida, Multifunctional Remote Laboratory for Education in Automatic Control: The CrAutoLab Experience, *IEEE Trans. on Industrial Electronics*, **55**(6), 2008, pp. 2376–2385.
9. M. Huba, M. Simunek, Modular Approach to Teaching PID Control, *IEEE Transactions on Industrial Electronics*, **54**(6), 2007, pp. 3112–3121.
10. J. Hashemi, N. Chandrashekar, and E.E. Anderson, Design and Development of an Interactive Web-based Environment for Measurement of Hardness in Metals: a Distance Learning Tool, *International Journal of Engineering Education*, **22**(5), 2006, pp. 993–1002.
11. A. Burns and A. Wellings, *RealTime Systems and Programming Languages*, 3rd edn, Addison-Wesley, UK, 2001.
12. TrueTime's home page, http://www.control.lth.se/truetime, Accessed 9 February 2010.
13. M. Ohlin, D. Henriksson and A. Cervin, *TrueTime 1.5 Reference Manual*, Department of Automatic Control, Lund University, Sweden, 2007.
14. The Matworks, *MATLAB 7 Getting Started Guide*, 1984–2008.
15. MATLAB's home page, http://www.mathworks.com, Accessed 9 February 2010.
16. Easy Java Simulations' home page, http://fem.um.es/Ejs, Accessed 9 February 2010.
17. F. Esquembre, Easy Java Simulations: A software tool to create scientific simulations in Java, *Comp. Phys. Comm.* **156**(2), 2004, pp. 199–204.
18. S. Dormido, F. Esquembre, G. Farias and J. Sánchez, Adding interactivity to existing Simulink models using Easy Java Simulations, *Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC*, Seville (Spain), 12–15 December 2005, pp. 4163–4168.
19. Scilab's home page, http://www.scilab.org/, Accessed 9 February 2010.

20. Sysquake's home page, http://www.calerga.com/, Accessed 9 February 2010.
21. G. Farias, F. Esquembre, J. Sánchez, S. Dormido, H. Vargas, S. Dormido-Canto, R. Dormido, N. Duro and M. Canto, Desarrollo de Laboratorios Virtuales, Interactivos y Remotos Utilizando Easy Java Simulations y Modelos Simulink, *12th Latin-American Congress on Automatic Control,* Salvador de Bahía (Brasil), 2006.
22. J. Sánchez, F. Esquembre, C. Martín, S. Dormido, S. Dormido-Canto, R.Dormido-Canto, R. Pastor and A. Urquía. Easy Java Simulations: An open source tool to develop interactive virtual laboratories using MATLAB/Simulink, *International Journal of Engineering Education: Special Issue on MATLAB and Simulink in Engineering Education*, **21**(5), 2005, pp. 798–813.
23. K. J. Åström and T. Hägglund, *Advanced PID Control*, ISA–The Instrumentation, Systems and Automation Society, Research Triangle Park, NC, 2005.

Gonzalo Farias received his BS degree in computer science from the University de la Frontera, Temuco, Chile, in 2001. He has been working, since 2005, in the Department of Computer Sciences and Automatic Control at National University for Distance Education (UNED), Madrid, Spain. His current research interests include control of dynamic system, and virtual and remote labs.

Karl-Erik Årzén received his Ph.D. in automatic control from the Lund Institute of Technology, Sweden, in 1987. He has been a professor in the Department of Automatic Control at Lund Institute of Technology since 2000. His research interests are real-time systems, real-time control, and programming languages for control applications.

Anton Cervin received the M.Sc. degree in Computer Science and Engineering in 1998 and the Ph.D. degree in Automatic Control in 2003, both from Lund University. He is currently an Associate Professor at the Department of Automatic Control at Lund University. His research interests include embedded and networked control systems, event-based control, real-time scheduling theory, and computer tools for analysis and simulation of controller timing.

Sebastián Dormido received his BS degree in physics from Complutense University, Madrid, Spain, in 1968 and his Ph.D. in science from the University of the Basque Country, Bilbao, Spain, in 1971. In 1981, he was appointed Professor of Control Engineering at National University for Distance Education (UNED), Madrid, Spain. His scientific activities include computer control of industrial processes, model-based predictive control, robust control, and model and simulations of continuous processes. He has authored or coauthored more than 150 technical papers in international journals and at conferences. Since 2002, he has promoted academic and industrial relations as President of the Spanish Association of Automatic Control CEA-IFAC (Comité Español de Automática-International Federation on Automatic Control).

Francisco Esquembre received his Ph.D. in mathematics from the University of Murcia, Murcia, Spain, in 1991. He has been employed at the University of Murcia since 1986, holding the position of Associate Professor since 1994. His academic expertise includes differential equations, dynamical system and numerical analysis. His research includes computer-assisted teaching and learning.