Defining and performing experiments in virtual laboratories

Francisco Esquembre
Departamento de Matemáticas, Universidad de Murcia, Campus de Espinardo, 30071 Murcia, Spain,
fem@um.es

Sebastián Dormido-Bencomo [1], Gonzalo Farias [2]

***Abstract* – Our work attempts to define and implement a generic experimentation language for conducting automatic experiments on existing simulations. Our objective is to be able to use a simulation, which may have been created independently, as a component in which students can perform experiments under different operating conditions. The experiments are first defined in a high-level language and then conducted on the simulation in an automatic way. The experimentation language should implement conditional execution of instructions that depend on the state of the simulation, running multiple copies of the simulation synchronously, and on-the-fly graphical comparison of results. This paper describes the elements required by such an experimentation language in order to provide the flexibility required for a wide variety of experiments. We also introduce our implementation of this language on the modeling tool Easy Java Simulations. Finally, we show examples of non-trivial experiments defined using this language and conducted on this software platform.**

Index Terms – Easy Java Simulations, Experimentation Language, Experiments, Simulations, Virtual Laboratories.

I. Introduction

The ultimate goal of building a simulation for a virtual laboratory is that of performing interesting experiments with the simulation. A typical definition of experiment states that "an experiment is the process of extracting data from a system by exerting it through its inputs" [1]. This definition needs to be made more general when our experimentation system is a computer simulation. Indeed, in a computer simulation, not only all its inputs and outputs are accessible, but modern modeling tools even allow for a direct control of the model so that its behavior can, to a certain extent, be changed in run-time. Traditionally, users of virtual laboratories are expected to perform experiments by interacting with the simulations' graphical user interface (GUI). But this frequently poses important limitations.

Consider, for instance, a computer simulation of the PI control of the level of a tank. An experiment for this simulation could consist of the following actions:

1. Set initial conditions.
2. Let the simulation evolve until the initial set point is reached with a 5% tolerance.
3. Increase the set point by 50%.
4. Let the system evolve until the exact moment when the level reaches the new set point with a 5% tolerance.
5. Compute the time elapsed in step 4.
6. Repeat steps 1 through 5 one hundred times with different sets of PI parameters.
7. Conduct an analysis on the results thus obtained.

This set of actions cannot be executed trivially, or in reasonable time, by a user interacting with the GUI. Some actions might be simply impossible without computer help. Instead, it would be preferable that users could count on a flexible experimentation language that allowed them to instruct the simulation to automatically run this experiment. This way, the virtual laboratory is treated as a complete system in which all variables are observable, and all variables and the simulation's execution itself are controllable.

Our work defines a standard set of actions that computer simulation experiments should implement. We do so by designing an API (Application Programming Interface) or set of instructions which simulations should conform to in order to provide standard experimentation capabilities. Some modeling or simulation environments already include scripting facilities that allow users to run certain types of experiments [2]. Among them ACSL, EcosimPro, and Dymola. For instance, Dymola's manual states that "…there is a script facility that makes it possible to load model libraries, set parameters, set start values, simulate, and plot variables by executing scripts". We are inspired by these previous experiences but have also added our own requirements to create a universal, full-fledged specification that provides more general and flexible features.

In order to test the viability of our language, we have implemented it using the modeling tool *Easy Java Simulations*, (Ejs). Ejs is a software tool that helps create interactive simulations in Java [5]. It has been designed specifically to be used by scientist without special programming skills, and has proven to simplify the creation of simulations for scientific and engineering purposes [11]. Simulations created with Ejs are complete Java applications or applets that can be distributed independently of Ejs. Our

[1] Sebastián Dormido-Bencomo, Universidad Nacional de Educación a Distancia, Spain, sdormido@dia.uned.es
[2] Gonzalo Farias, Universidad Nacional de Educación a Distancia, Spain, gfarias@bec.uned.es

goal is that these simulations implement our experimentation language.

The paper is organized as follows. Section II lists the requirements for our experimentation language. Section III discusses the implementation of the language done using Easy Java Simulations. Section IV shows two examples that use our experimentation language in practice. Finally, Section V discusses the results and describes further work.

II. Elements of the Language

Our objective is to be able to control every aspect of a simulation as if it were a completely observable and controllable component. Our experimentation language should then contain the following categories of elements, or instructions, in its API:

A. Elements to run one or more instances of a simulation.
B. Elements to access variables and routines.
C. Elements to specify algorithms.
D. Elements to control the execution of the simulation.
E. Elements for user input.
F. Elements to allow for comparison of results.

We now discuss each of these categories in more detail.

A. Elements to run one or more instances of a simulation

Users may want to run different simulations, or several instances of the same simulation, at the same time in order to compare results among simulations. The API should then provide an instruction to launch any simulation users have access to, returning a unique identifier for it.

Users should also be able to specify whether they want the running simulations to execute either synchronously or asynchronously. Synchronized simulations advance (step) through their evolution cycle at the same pace. In particular, if the simulations use the same increment of time for each step, their internal time will remain synchronized.

B. Elements to access variables and routines

Users need to be able to read and to set the value of the variables of the model of a simulation at any time. This can only be restricted if the simulation designer has declared some of the model variables as non-accessible (private). The same principle applies to routines or functions (methods) that the simulation defines. Users should be able to easily obtain information about available variables and methods.

C. Elements to specify algorithms

The API should allow users to perform any required computation. These computations can make use of variables and methods from the simulation model, as well as of additional ad-hoc (local) variables defined by users. The language must provide for standard algorithmic constructions to allow users to write complex algorithms, if required.

D. Elements to control the execution of the simulation

Users may want to control the simulation execution. This includes not only standard play and pause instructions that start/stop the simulation, but also instructions to run the simulation until a given condition is met, such as (in human language): “run the simulation until the level of the tank is greater than 10”. The experimentation environment should then be able to pass over the control of the computer resources to the running simulation and wait until the simulation meets the given criteria and pauses, thus giving control back to the experiment. Another feature required is the possibility of planning events in the future, such as: “run the simulation increasing the set point by 50% when t = 10”.

E. Elements for user input

In occasions, partial results of the experiment may require user input. Elements in this category should allow displaying messages or asking users to enter one or more numeric values, choose a given option out of several offered, or confirm an action.

F. Elements to allow for comparison of results

In experiments where a simulation is run several times, each under different conditions, users will most likely want to store intermediate or output results in order to compare them at the end of the different runs of the simulation. Hence, the API should provide some kind of memory where to store, and later retrieve, these values. Also, the API should provide a means to visually compare output data from a simulation produced in form of a graph. For instance, users can be interested in comparing the plots of the evolution in time of the response of a PI control under different tuning parameters.

III. Implementation

We chose Easy Java Simulations for our implementation because it offers several appropriated characteristics. Ejs falls into the category of code generators, which makes it possible to use for our API all the constructions provided by a standard programming language. The fact that Ejs is based on Java has also been crucial in our work because it helps manage several instances of a simulation, or address compound objects (such as graphs) in them, in an object-oriented way. Finally, users of Ejs can easily inspect, understand, and, if necessary even modify, other people’s simulations, which greatly increases their observability and controllability.

Easy Java Simulations is a modeling and authoring tool that allows specifying the model and the view for a simulation at a very high-level of abstraction. We have added to Ejs the possibility of defining experiments for existing simulations by loading the XML file that describes the simulation (which may have been created by another person) and adding pages defining experiments for it. When the simulation is re-generated, it adds to its standard menu an entry for each of the experiments thus defined. Users simply select the experiment as one menu option. When running the simulation as an applet, the experiments can also be accessed using hyperlinks embedded in the HTML page that contains the simulation. This possibility provides a way to include, in a very natural way, the execution of experiments on the simulation in curricular material developed in HTML form.

To implement our experimentation language, we have added to Easy Java Simulations new predefined methods that

provide the necessary functionality. We now describe how we implemented the elements in each of the categories of the API.

A. Elements to run one or more instances of a simulation

Our API provides two instructions to create a running instance of a simulation:

```
public Model runSimulation ( );
public Model runSimulation (String classname);
```

These are instance methods of a predefined object called _simulation, which points to the simulation itself. The first method creates and runs a copy of the simulation from which the experiment is started. The second method creates a copy of the simulation with the given class name. Every Java simulation is an object of a given class and several classes can be packaged together in compressed archives called JAR files. Users can instantiate any simulation which is in the same JAR file as the original simulation or in any other JAR file included in the simulation's class path. Ejs simulations can add JAR files to their class path using the *Additional Libraries* field in the model editor of Ejs.

Simulations created using any of these two methods appear automatically on the computer screen and are by default synchronized with (we call them subordinates of) the original one. Subordinates of a simulation can be freed (made to run asynchronously) using the _simulation instance method:

```
public void freeSimulation (Model subordinate);
```

Finally, subordinate simulations can be disposed of by calling one of the following instance methods of _simulation:

```
public void killSimulation (Model subordinate);
public void killAllSimulations ( );
```

Although seldom required, a single simulation can create more than one subordinate simulation, which can in turn create their own subordinate simulations. All subordinate simulations in the same family are, by default, synchronized. Exiting any of them, exits all the simulations in the family.

B. Elements to access variables and routines

Experiments are created and run as part of the model of a simulation. This gives them direct access to the model's variables and methods. Both versions of the runSimulation method described above return an object of the corresponding model class, which is an implementation of the generic Java interface *org.opensourcephysics.ejs.Model*, included by default in every Ejs simulation's JAR file. Users need to typecast this object into a local variable of the correct type in order to access the model's public variables and methods. The standard object-oriented "dot" mechanism of Java can then be used to address any variable or method in the simulation model.

As an example, suppose that we are running an experiment from a simulation whose model is of the class MySimModel, and which has a variable called x and a method called action. The experiment can then use constructions of the form:

```
// Create a subordinate instance of this simulation
MySimModel sub = (MySimModel) _simulation.runSimulation( );
x = 1.0;          // Sets the x variable of this simulation
action( );        // Invokes the action method of this simulation
sub.x = 0.0;   // Sets the x variable of the subordinate
sub.action( ); // Invokes the subordinate's action method
_play( );         // Plays both simulations synchronously
```

C. Elements to specify algorithms

We used the fact that Ejs is a code generator tool to allow users to write any valid Java construction in the algorithms of the experiments. These constructions can, and typically do, make use of the methods defined in our experimentation API. When the simulation is generated, Ejs compiles the Java code for the experiments together with the rest of the simulation model.

D. Elements to control the execution of the simulation

Ejs already included a set of predefined methods that allow users to control the execution of a simulation. These methods are described in the Ejs manual and feature:

```
void _play( );     // Plays the simulation
void _pause( );  // Pauses the simulation
void _step( );     // Advances by one time step
void _reset( );    // Completely resets the simulation
```

Because experiments are run in a Java thread different to that of the simulation itself, our API has extended this set with the method:

```
void _playAndWait ( );
```

which has a similar effect to _play in the original set, but delays the execution of code after this instruction until the simulation pauses.

A simulation can be paused by either user interaction, an invocation of the _pause method included in the original simulation, or by using one of the following new instructions:

```
void _scheduleCondition (String conditionName);
void _scheduleEvent (String eventName);
```

These two methods introduce the possibility of executing code whenever a given condition is satisfied. This code can be used to simply pause the simulation or to execute other more complex actions. The parameter of both instructions refers to an instance of one of the new constructions called *scheduled condition* and *scheduled event*, respectively, which can be defined using a special editor provided by Ejs. Both constructions consist of two methods each. The first method determines whether a given condition is satisfied by the model state. The second method defines a user-defined action that will be invoked when this condition is met.

There are some differences between both constructions. Scheduled conditions are determined by a method returning a boolean value, which is tested after every simulation step. If the method returns a true value, the corresponding action is executed. Scheduled events are associated to any of the systems of ordinary differential equations (ODEs) defined by the model as part of its evolution algorithm, and are triggered by the change in sign of a positive function of the variables involved in that system of ODEs. When the function returns

a negative value, the simulation detects the event, goes back in time to find the exact instant in time when the function crossed zero, and applies the event action at that instant. In this sense, scheduling an event is similar to adding new events to the original system of ODEs in runtime. Differently to normal events, though, scheduled events (and scheduled conditions, as well) disable themselves automatically once they take place.

E. Elements for user input

Our API provides a new predefined _input object that implements a simple mechanism for user input during an experiment. This object has the following instance methods:

```
int confirmMessage (String message, int type);
int selectOption (String message, String options);
boolean inputVariables (String message, String variables);
```

The first of these input methods is used to display a message the user must acknowledge or prompt the user to confirm a yes/no type question. The second method is used to request the user to select an option out of several possible ones. The third method displays a table in which the user needs to input a value for each of the variables specified by a comma separated list of names. These names create internal variables in the _input object whose values can be retrieved using the getter methods:

```
boolean getBoolean (String variable);
int getInt (String variable);
double getDouble (String variable);
String getString (String variable)
Object getObject (String variable);
```

The last of these getter methods can be used to retrieve arrays or other Java objects with a textual representation. The variables can be assigned values previously to user input using the setter methods:

```
void setValue (String variable, boolean value);
void setValue (String variable, int value);
void setValue (String variable, double value);
void setValue (String variable, Object value);
```

These values will then be displayed as default values by the input table. Differently to the _memory object discussed below, variables in the _input object are cleared at the beginning of each experiment.

F. Elements to allow for comparison of results

The API also provides a new predefined object called _memory, which can be used to store and retrieve data while running an experiment or across different experiments. The memory has the same setter and getter methods as the _input object, if only its variables remain accessible from experiment to experiment unless its instance method:

```
void clear ( );
```

is explicitly invoked. Data in the memory can be used for post-experiments analysis.

Comparing graphs is possible thanks to the object oriented nature of Java. Any graphical element in the simulation view is a public object whose methods can be accessed just like any other method of the simulation. We have added a new instruction to our API that allows cutting and pasting drawables elements from one graphic panel to another:

```
void reparentDrawable(String childName,
                      ControlElement newParent);
```

Drawables is the generic name we use to refer to objects which draw on graphic panels. ControlElement is the parent class of all graphic elements in the view of a simulation created with Ejs. This method can be used to effectively display a drawable object which is originally part of, and receives data from, one simulation into the drawing panel of the other simulation. See Experiment II below for an example of use.

IV. EXAMPLES

We show in this section two examples of experiments created for a simulation of the PI control of the level of a tank. The simulation's typical behavior for default Kp and Ti values of the PI controller is shown in Figure 1.

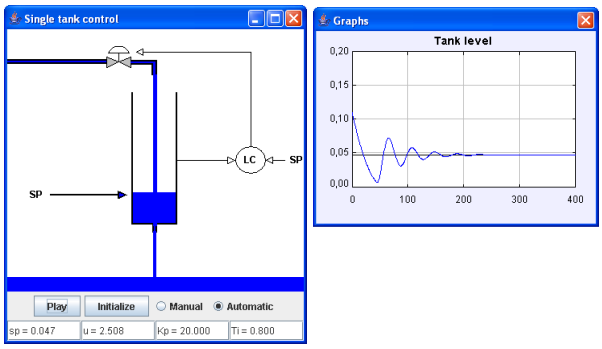

FIGURE 1
TYPICAL RESPONSE OF THE SINGLE TANK SMULATION.

The dynamics of this single tank simulation is determined by the *Dynamics* page of ordinary differential equation (ODE) shown in Figure 2.

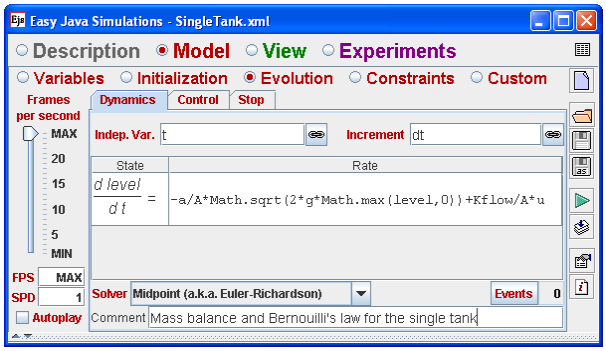

FIGURE 2
DYNAMIC EQUATIONS OF THE SIMPLE TANK SYSTEM..

The control signal u is computed in the second page of the evolution of the model using the following code:

```
if (automaticMode) {
  // P + I action
  u = Kp*(setPoint - level) + integral;
  if (u<0) u = 0;
  // Update integral action
  integral = integral + Kp*dt/Ti * (setPoint - level);
}
```

which implements a digital PI action.

EXPERIMENT I. Executing a scheduled event

We now want to define a very simple experiment consisting in doubling the set point when the time equals 200 seconds. For this, we first define a scheduled event in the dedicated panel of Ejs, as shown in Figure 3.

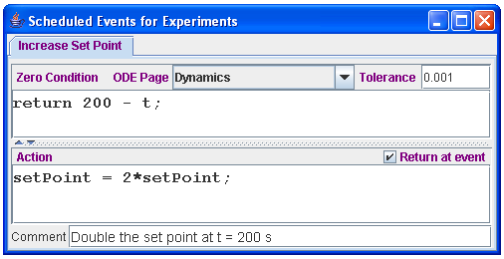

FIGURE 3
EDITOR FOR SCHEDULED EVENTS.

As the code in the figure shows, the event is triggered when the time exceeds the time planned for the event (200 seconds). When the simulation detects the crossing condition, it goes back in time through the *Dynamics* ODE to find the exact state at instant t = 200. It then executes the event action which doubles the set point. Notice that events defined using this editor are independent of events the simulation may have defined as part of its model and are not activated until explicitly set by an _scheduleEvent instruction. As mentioned above, scheduled events are automatically removed from the ODE list of events once they take place.

We then turn to the panel for experiments in Ejs' interface and create a new page with the code displayed in Figure 4.

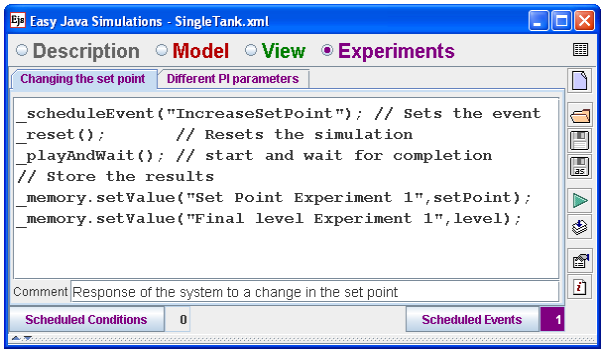

FIGURE 4
DEFINITION OF EXPERIMENT I IN EJS.

If we now run the simulation the popup menu of the main drawing panel includes an entry for the experiment. See Figure 5. (Experiment II defined in the next subsection is also displayed.)

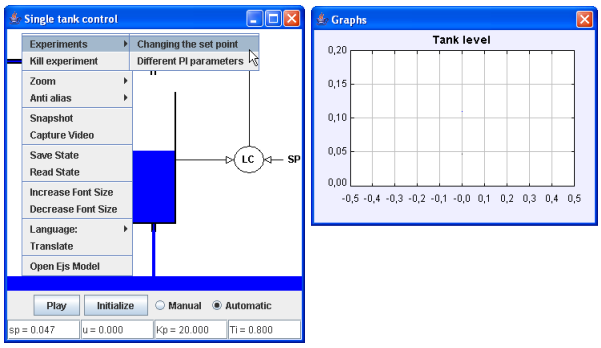

FIGURE 5
RUNNING EXPERIMENT I FROM THE SIMULATION INTERFACE.

Selecting this experiment in the menu produces the results of Figure 6.

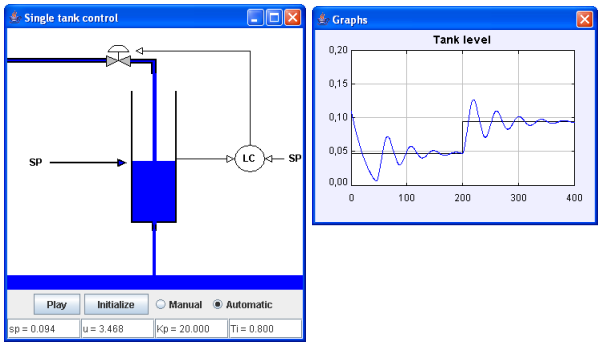

FIGURE 6
OUTPUT OF EXPERIMENT I.

When the simulation finally stops, the _memory object stores the values of the set point and the level. These values can be used for further studies.

EXPERIMENT II. Comparing graphic outputs

We now want to compare the responses of the PI control with different Kp and Ti parameters. A simplistic solution would be to run the simulation by hand twice, once for each set of parameters, take snapshots of the evolution graphs, and then compare them looking at each graph side by side. A better procedure, though, is to conduct an experiment that automatically creates a second copy of the simulation, changes its parameters, and then runs both simulations synchronously, displaying the graph of their responses in the same plot. This is what the following, more elaborate, experiment does:

```
_reset(); // Resets the simulaton
// Creates a subordinate simulation
SingleTank subordinate = (SingleTank) _simulation.runSimulation();
subordinate.Kp = 30;  // Sets the subordinate's Kp
subordinate.Ti = 1.0;  // Sets the subordinate's Ti
java.awt.Color color = java.awt.Color.RED; // Chooses a color
// Changes the color of the subordinate's level trace
subordinate._view.levelTrace.getStyle().setEdgeColor(color);
// Reparents the subordinate's level trace into the plotting panel
subordinate._view.reparentDrawable("levelTrace",
                                   _view.getElement("plottingPanel"));
subordinate._view.dispose(); // Hides the subordinate's view
_play(); // plays both simulations
```

The output of this experiment is shown in Figure 7.

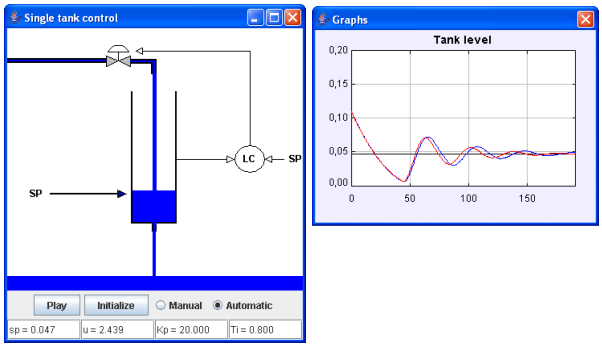

FIGURE 7
OUTPUT OF EXPERIMENT II.

IV. DISCUSSION OF RESULTS

We are in the process of using our implementation to test our initial design creating different types of experiments of practical use in teaching Automatic Control and other topics (such as Physics). Our initial results show our implementation is both simple and flexible, allowing us a great deal of control of the running simulation. The object-oriented nature of Java has been crucial in making our implementation very natural. The way Easy Java Simulations lets users inspect simulations created by other people and access all its variables and methods is also of great importance to reduce to a real minimum the documentation work required by the author of the original simulation.

In a more general context, we think our API can be the basis for the definition of a standard experimentation language to which other modeling and simulations tools could adhere to. This is our goal in the coming future.

The current, experimental version of Ejs that supports the features described in this paper can be downloaded from http://www.um.es/fem/publications/2007/Ejs070507.zip. The experiments of Section IV are in the **SingleTank.xml** file in the **_examples/Experiments** directory.

ACKNOWLEDGMENT

This work was supported by the Spanish CICYT under grant DPI2004-01804 and the Autonomous Region of Madrid CAM under grant S-0505/DPI-0391.

REFERENCES

[1] Cellier, F, "Computer System Modeling", *Springer-Verlag*, 1991.

[2] Elmqvist, H, Mattsson, S. E., Otter, M. "MODELICA — the new object-oriented modelling language", *The 12th European Simulation Multiconference*, 1998, pp 16—19.

[3] Brück, D, Elmqvist, H, Mattsson S.E. , Olsson H. "Dymola for Multi-Engineering Modeling and Simulation". *Proceedings of the 2nd International Modelica Conference*, 2002, pp 55-1—55-8

[4] Fritzson, O, Gunnarsson, J, Jirstand, M. "MathModelica. An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming". *Proceedings of the 2nd International Modelica Conference*, 2002, pp 41—54.

[5] Esquembre, F. "Easy Java Simulations: a software tool to create scientific simulations in Java", *Comp. Phys. Comm*, Vol, 156, 2004, 199-204.

[6] Esquembre, F. "Easy Java Simulations' web site", http://www.um.es/fem/Ejs.

[7] Dormido, S.; Esquembre, F. "The Quadruple-Tank Process: An Interactive Tool for Control Education", *Proceedings of the European Control Conference*, 2003.

[8] S. Dormido. "Control learning: Present and future", *Annual Reviews in Control*, vol. 28, 2004, pp. 115-136.

[9] Dormido, S.; Martín, C.; Pastor, R.; Sánchez, J.; Esquembre, F. "Magnetic Levitation System: A Virtual Lab in Easy Java Simulation", *Proceedings of the American Control Conference*, 2004.

[10] Martín, C.; Urquia, A.; Sánchez, J.; Dormido, S.; Esquembre, F.; Guzmán, J. L.; Berenguel, M. "Interactive simulation of object-oriented hybrid models, by combined use of EJS, Matlab/Simulink and Modelica/Dymola", *Proceedings of the 18th European Simulation Multiconference*, 2004, pp. 210-215.

[11] Sánchez, J, Dormido, S., Esquembre, F. "The learning of control concepts using interactive tools". *Comp. App. Sci. Eng.* Vol .13, 2005, pp 84-98.

[12] J. Sánchez, F. Esquembre, C. Martín, S. Dormido, S. Dormido-Canto, R.Dormido-Canto, R. Pastor, A. Urquía. "Easy Java Simulations: An open source tool to develop interactive virtual laboratories using Matlab/Simulink", *The International Journal of Engineering Education: Special Issue on MATLAB and Simulink in Engineering Education*, vol. 21, nº 5, 2005, pp. 798-813.

[13] Dormido, S.; Esquembre, F.; Farias, G.; Sánchez, J. "Adding interactivity to existing Simulink models using Easy Java Simulations", *Proceedings of the Conference Decision and Control-European Control Conference*, 2005.

[14] R. Dormido; H. Vargas; N. Duro; J. Sánchez; S. Dormido-Canto; G. Farias; F. Esquembre; S. Dormido. "Development of a web-based control laboratory for automation technicians: The three-tank system", *IEEE Trans on Education*, (accepted), will appear in nov. 2007