AN APPROACH TO VIRTUAL-LAB IMPLEMENTATION USING MODELICA

Carla Martin, Alfonso Urquia and Sebastian Dormido
Dept. Informática y Automática, ETS Ingeniería Informática, UNED
Juan del Rosal 16, 28040 Madrid, Spain
E-mail: {carla, aurquia, sdormido}@dia.uned.es

KEYWORDS
Interactive simulation, object-oriented modeling, hybrid-models, education.

ABSTRACT

An approach to the implementation of virtual-labs is discussed in this manuscript. It allows describing the view (i.e., the user-to-model interface) and the model of the virtual-lab using Modelica language. To achieve this goal, the following two tasks have been completed: (1) a methodology to transform any Modelica model into a formulation suitable for interactive simulation has been proposed; and (2) *VirtualLabBuilder* Modelica library has been designed and programmed.
VirtualLabBuilder library includes Modelica models implementing a set of graphic interactive elements, such as containers, animated geometric shapes (polygon and oval) and interactive controls (slider and radio-button). These models allow the user: (1) to define the interactive graphic elements composing the virtual-lab view; and (2) to link the model variables with the geometric properties of these graphic elements.
The structure, capabilities and use of *VirtualLabBuilder* library are discussed in this manuscript. The library use is illustrated by means of a simple example. Finally, *VirtualLabBuilder* is used to implement the virtual-lab of the quadruple-tank process.

INTRODUCTION

A virtual-lab is a distributed environment of simulation and animation tools, intended to perform the interactive simulation of a mathematical model. Virtual-labs provide a flexible and user-friendly method to define the experiments performed on the model. In particular, interactive virtual-labs are effective pedagogical resources, well suited for distance education.

Typically, the virtual-lab definition includes the following two parts: the *model* and the *view*. The *view* is the user-to-model interface. It is intended to provide a visual representation of the model dynamic behavior and to facilitate the user's interactive actions on the model.

The graphical properties of the *view* elements are linked to the model variables, producing a bi-directional flow of information between the *view* and the *model*. Any change of a model variable value is automatically displayed by the *view*. Reciprocally, any user interaction with the *view* automatically modifies the value of the corresponding model variable.

Modelica (http://www.modelica.org/) is an object-oriented modeling language that facilitates the physical modeling paradigm (Åström et al. 1998). It supports a declarative (non-causal) description of the model, which permits better reuse of the models. As a consequence, the use of Modelica reduces considerably the modeling effort. However, neither Modelica language nor Modelica simulation environments (e.g., Dymola (Dynasim 2006)) support interactive simulation. As a consequence, extending Modelica capabilities in order to facilitate interactive simulation is an open research field.

Previous work on this topic includes (Engelson 2000; Martin et al. 2004; Martin et al. 2005a; Martin et al. 2005b). In particular:

- The combined use of Modelica/Dymola, Matlab and Easy Java Simulations (Esquembre 2004; http://www.um.es/fem/Ejs) is proposed in (Martin et al. 2004; Martin et al. 2005a; Martin et al. 2005b). This approach allows the implementation of virtual-labs with *runtime interactivity*. The user is allowed to perform actions on the model during the simulation run. He can change the value of the model inputs, parameters and state variables, perceiving instantly how these changes affect to the model dynamic. An arbitrary number of actions can be made on the model during a given simulation run.
- The combined use of Modelica/Dymola and Sysquake (http://www.calerga.com) is proposed in (Martin et al. 2005a). This approach facilitates the implementation of virtual-labs with *batch interactivity*. The user's action triggers the start of the simulation, which is run to completion. During the simulation run, the user is not allowed to interact with the model. Once the simulation run is finished, the results are displayed and a new user's action on the model is allowed.

The goal of the work discussed in this manuscript is the programming of a Modelica library supporting the implementation of virtual-labs with *runtime interactivity*. This novel Modelica library, named *VirtualLabBuilder*, allows the user to define the model and the view of the virtual-lab, and the link between them, using only Modelica.

The architecture and use of *VirtualLabBuilder* library is described in the following sections, and *VirtualLabBuilder* is used to implement the virtual-lab of the quadruple-tank process.

DESCRIPTION OF THE PROPOSED APPROACH

The virtual-lab description is composed of the model description and the view description.

a) The *virtual-lab model* has to be written in Modelica language, according to the methodology proposed in (Martin et al. 2005a). Essentially, this approach imposes that all the interactive variables have to be state variables. In particular, in order to allow interactive changes in the value of model parameters and input variables, they have to be written as zero-derivative state variables. This methodology can be applied to any Modelica model.

b) The Modelica description of the *virtual-lab view* (i.e., the view class) has to be a subclass of the `PartialView` Modelica class. `PartialView` is included in the *VirtualLabBuilder* library and it contains the code required to perform the model-to-view communication. This code is valid for any model and view descriptions, and the user only needs to set the length of the model-to-view communication interval.
In addition, the user has to include within the view class: (1) the required instantiations of the graphic interactive elements composing the virtual-lab view; and (2) the connection among these elements. The *VirtualLabBuilder* library contains a set of ready-to-use graphic elements. The connection among these elements determines their layout in the virtual-lab view. Dymola GUI allows defining in a drag-and-drop way the instantiation of these elements and connecting them using the mouse.

c) The Modelica description of the *virtual-lab* has to be an instance of `VirtualLab` class. This Modelica class is included in *VirtualLabBuilder* library. The user has to provide the name of the model class and the view class. Also, the user has to specify how the geometric properties of the view elements are linked to the model variables.

The virtual-lab description, obtained as discussed in c), is translated using Dymola and run. As a part of the model initialization (i.e., the calculations performed to find the initial value of the model variables), the initial sections of the interactive graphic objects and of the `PartialView` class are executed. These initial sections contain calls to Modelica functions, which encapsulate calls to external C-functions. These C-functions are Java-code generators.

As a result, during the model initialization, the Java code of the virtual-lab view is automatically generated, compiled and bundled into a single jar file. Also, the communication procedure between the model and the view is set up. This communication is based on client-server architecture: the C-program generated by Dymola is the server and the Java program automatically generated during the model initialization is the client.

Once the jar file has been created, it has to be executed by the user. As a result, the initial layout of the virtual-lab view is displayed and the client-server communication is established. Then, the model simulation starts.

During the simulation run, there is a bi-directional flow of information between the model and the view. The communication is as follows. Every communication interval:

- The model simulation (i.e., the server) sends to the view (i.e., the client) the data required to refresh the view.
- The view sends to the model simulation the new value of the variables modified due to the user's interactive action.

VirtualLabBuilder ARCHITECTURE

VirtualLabBuilder library is composed of the following five packages (see Figure 1a).

- The `ViewModel` package contains the `PartialView` and the `VirtualLab` classes.
- The `ViewElements` package contains the graphic interactive elements that the user can employ to compose the view. The content of this package is shown in Figure 1b and it will be described in the next section.
- The `Interfaces` package contains the interfaces (i.e., connectors) of the graphic interactive elements.
- The `Functions` package contains the Modelica functions which encapsulate calls to external C-functions. As discussed in the previous section, these C-functions are Java-code generators.
- The `TypesDef` package contains the definition of several types of variables. These types are intended to be used for defining some properties of the graphic interactive elements (such as color, layout, etc.).

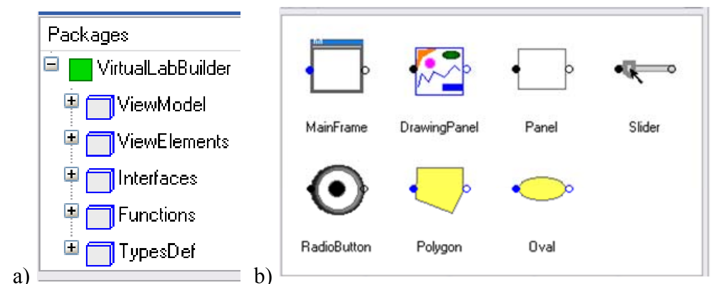

Figure 1: a) Packages of the *VirtualLabBuilder* Library; and b) Classes within the `ViewElements` Package

GRAPHIC ELEMENTS

The `ViewElements` package contains the graphic elements that can be used to define the view. These elements (see Figure 1b) can be classified into the following three categories.

- *Containers* (`MainFrame`, `Panel` and `DrawingPanel` classes). These graphic elements can host other graphic elements. The properties of these elements are set in the view definition and they can not be modified during the simulation run.

- *Drawables* (`Polygon` and `Oval` classes). These elements can be used to build an animated schematic representation of the system. The variables setting the geometric properties of these elements (position, size, etc.) can be linked to model variables.
- *Interactive controls* (`Slider` and `RadioButton` classes). Model variables can be linked to the variables defining the states of the interactive control elements. This allows the user to change the value of these model variables during the simulation run.

Drawable elements and interactive controls implement the information flow between the model and the view of the virtual-lab. The simulated value of the model variables modifies the properties of the drawable elements (i.e., model-to-view information flow). The user's interactive action on the interactive controls modifies the value of the model variables (i.e., view-to-model information flow). The properties of the graphic elements are discussed next.

Containers

`MainFrame` class creates a window where containers and interactive controls can be placed. The view can only contain one `MainFrame` object. This class has the following parameters:

- *width* and *height*: width and height of the window in pixels.
- *title*: text shown in the top part of the window.
- *layoutPolicy*: layout policy of the element. It sets where the elements placed within the window are located. Possible values are BorderLayout, GridLayout, HorizontalBox, VerticalBox and FlowLayout.

`Panel` class creates a panel where containers and interactive controls can be placed. This component is similar to `MainFrame`, however there is a difference: the view can contain more than one panel.

`DrawingPanel` class creates a two-dimensional container that only can contain drawable objects (i.e., `Polygon` and `Oval` objects). It represents a rectangular region of the plane which is defined by means of two points: (*XMin*, *YMin*) and (*Xmax*, *YMax*). The coordinates of these two points (i.e., the value of *XMin*, *XMax*, *YMin* and *YMax*) are parameters of the class whose value can be set by the user.

Drawables

`Oval` class draws an oval. The center position and the lengths of the axes can be linked to the model variables. The class has the following parameters:

- *lineColor*: color of the line.
- *fillColor*: color of the filling.
- *filled*: indicates whether the oval is filled or empty.
- *intCenter*: indicates whether the oval position changes during the simulation or remains constant.
- *intAxes*: indicates whether the oval shape changes during the simulation or remains constant.

`Polygon` class draws a polygonal curve specified by the coordinates of its vertexes points. The class has the following parameters:

- *lineColor*: color of the line.
- *fillColor*: color of the filling.
- *filled*: indicates whether the oval is filled or empty.
- *intVertexesX*[:]: array that indicates whether the horizontal position of each polygon's point changes during the simulation or remains constant.
- *intVertexesY*[:]: array that indicates whether the vertical position of each polygon's point changes during the simulation or remains constant.

Interactive Controls

`Slider` class creates a slider. This class has the following parameters:

- *position*: slider position inside the container object.
- *stringFormat*: format used to display the value.
- *tickNumber*: number of ticks.
- *tickFormat*: tick format.
- *enable*: allows enabling/disabling the object.
- *initialValue*: initial value of the slider variable.

`RadioButton` class creates a radio-button control.

CONNECTING THE GRAPHIC ELEMENTS

The interfaces of the *container*, the *drawable* and the *interactive control* classes are composed of two connectors: one filled and one empty (see Figure 1b). The user must observe the following three rules when connecting the graphic elements:

1. The connection between two components must be established by connecting the filled connector of one component with the empty connector of the other component.
2. Each filled connector must be connected to one and only one empty connector.
3. Empty connectors can be left unconnected. If they are connected, the allowed number of filled connectors connected to a given empty connector depends on the type of the graphic elements. This number is shown in Figure 2.

empty connector \ filled connector	MainFrame	DrawingPanel	Panel	Interactive Controls	Drawable
MainFrame		$\geq 1^{(*)}$	$\geq 1^{(*)}$	$\geq 1^{(*)}$	
DrawingPanel					1
Panel		$\geq 1^{(*)}$	$\geq 1^{(*)}$	$\geq 1^{(*)}$	
Interactive Controls				1	
Drawable					1

Figure 2: Allowed Number of Connections

(*): If the layout policy of the element is BorderLayout then ≥ 1 else 1.

LIBRARY USE

The steps to compose the Modelica description of a virtual-lab are described below. They are illustrated by means of a simple example: the implementation of the virtual-lab of the tank model shown in Figure 3.

*Model definition***.** The voltage applied to the pump (v) is an input variable. The cross-sections of the tank (A) and the outlet hole (a), the pump parameter (k) and the gravitational acceleration (g) are time-independent properties of the physical system. The physical parameters A and a, and the input variable v can be modified by the user action during the interactive simulation. Interactive parameters and input variables have been declared in the model as Real variables with zero time-derivative. It is assumed that the Modelica class of the tank has been programmed and it is called `PhysicalModel`.

View definition. Create a new class extending the `PartialView` class and call it `ViewModel`. Set the value of the model-to-view communication interval, which is a parameter (called `Tcom`) of the `PartialView` class. The `PartialView` class contains one graphic element: `root`. The object of the `MainFrame` class must be connected to this element. Add the `MainFrame` object and the other required graphic objects to the `ViewModel` class. Connect the graphic objects. The diagram of the obtained class is shown in Figure 4. Set the value of the graphic object parameters.

Virtual-lab definition. Create a new object of the `VirtualLab` class. This class contains two parameters: the class of view (`ViewI`) and the class of the model (`ModelI`). Set the values of these parameters to `ViewModel` and `PhysicalModel` respectively. Finally, write the equations required to link the view variables with the model variables.

Virtual-lab run. Translate (for instance, using Dymola) and simulate the object created previously. During the initialization calculations, the jar file is automatically generated. Execute the jar file. The virtual-lab view is displayed (see Figure 5a) and the interactive simulation of the virtual-lab starts. The time evolution of the model variables can be plot using Dymola. It is shown in Figure 5b for some selected variables. The discontinuous changes are due to user's interactive actions.

CASE STUDY: IMPLEMENTATION OF THE QUADRUPLE-TANK PROCESS VIRTUAL-LAB

The quadruple-tank process is represented in Figure 6. It can be used to explain different aspects of the multivariable control theory (Johansson 2000; Dormido and Esquembre 2003). The goal is to control the level of the two lower tanks with two pumps. The virtual-lab has been implemented as described in the previous section. It supports interactive changes in the liquid levels, the pump input voltages and the valve settings. The Modelica description of the view is shown in Figure 7 and the virtual-lab view in Figure 8. The time evolution of the liquid levels can be plot using Dymola (see Figure 9).

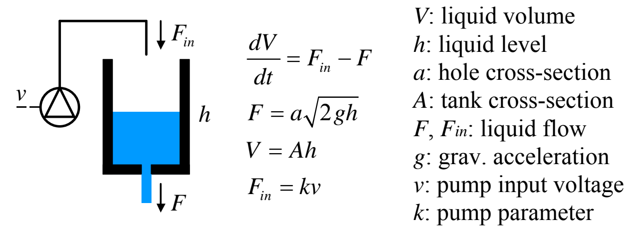

Figure 3: Model of a Tank System

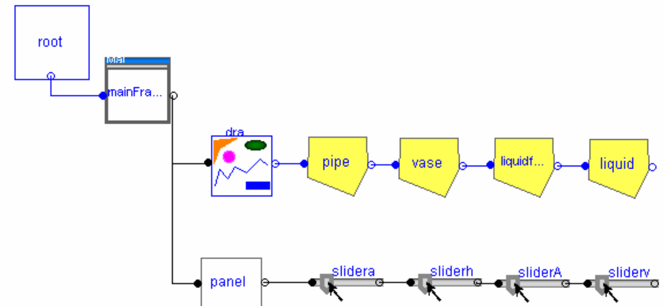

Figure 4: Diagram of the Modelica Description of the View

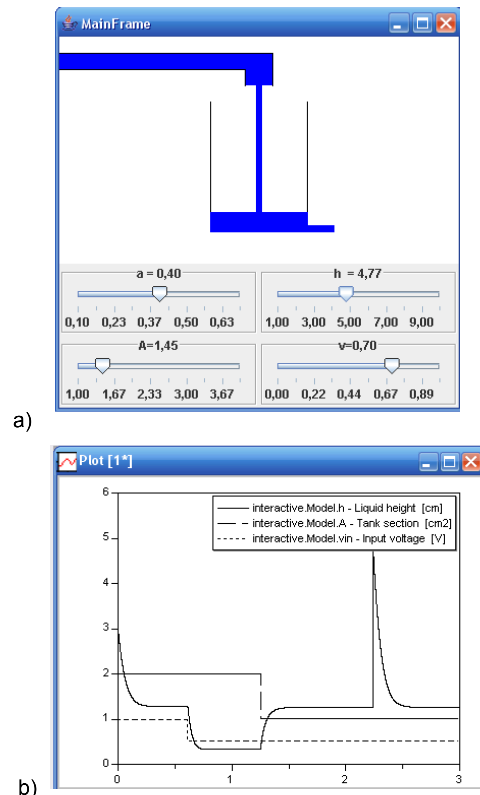

Figure 5: a) Virtual-lab View; b) Variable Plots

CONCLUSIONS

A novel approach to the virtual-lab implementation using Modelica language has been proposed. In order to put it into practice, two tasks have been completed: (1) the proposal of a modeling methodology intended to transform any Modelica model into a description suitable for interactive simulation; and (2) the design and programming of a Modelica library supporting the description of the virtual-lab view and the bi-directional communication between the model and the view. The purpose, structure and use of this Modelica library, called *VirtualLabBuilder*, have been discussed and its use has been illustrated by means of two examples.

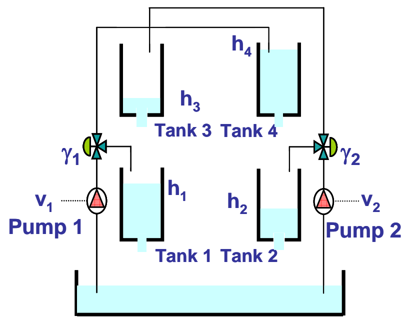

Figure 6: The Quadruple-Tank Process

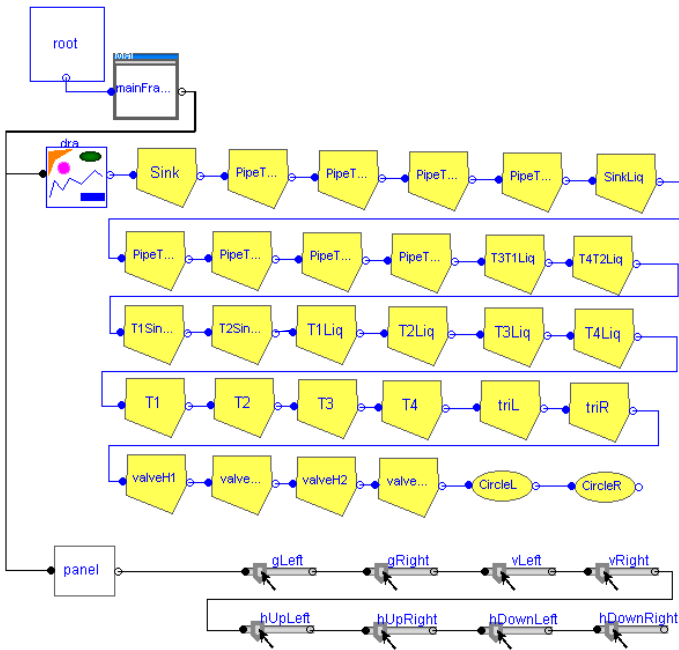

Figure 7: Diagram of the Modelica Description of the View

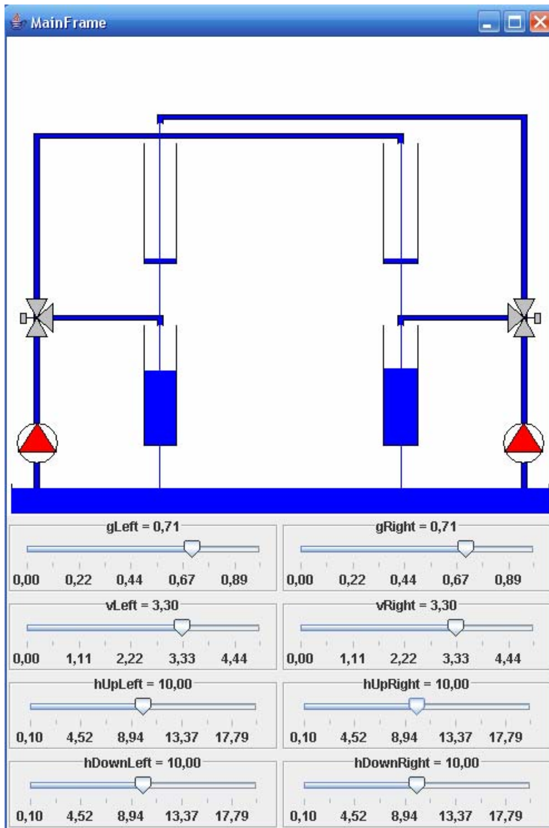

Figure 8: Virtual-Lab View

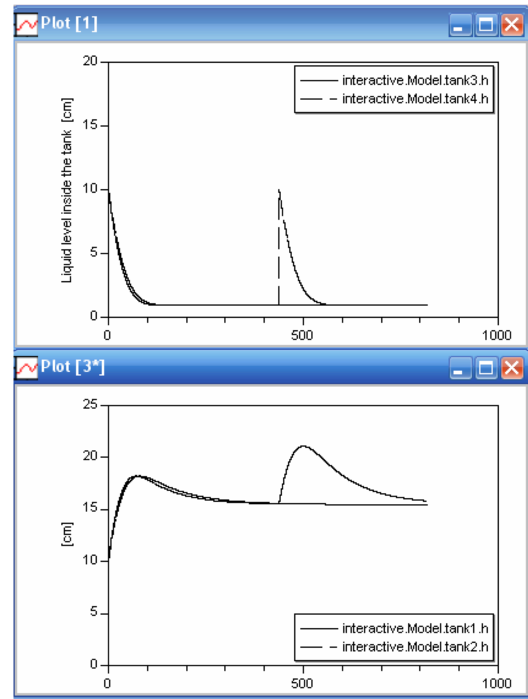

Figure 9: Plots of Selected Process Variables

REFERENCES

Åström K.; H. Elmqvist and S. E. Mattsson. 1998. "Evolution of Continuous-Time Modeling and Simulation". In *Proceedings of the 12th European Simulation Multiconference* (Manchester, UK).

Dormido, S. and F. Esquembre. 2003. "The Quadruple-Tank Process: An Interactive Tool for Control Education", *In Proceedings of the 2003 European Control Conference*, (Cambridge, UK).

Dynasim. 2006. "Dymola. User's Manual". Dynasim AB. Lund, Sweden.

Engelson V. 2000. "Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing". Ph. D. Thesis, Dept. of Computer and Information Science, Linköping University, Sweden. Dissertation No. 627.

Esquembre F. 2004. "Easy Java Simulations: a Software Tool to Create Scientific Simulations in Java". *In Computer Physics Communications*, Vol. 156, 199-204.

Johansson K.H. 2000. "The Quadruple-Tank Process: A Multivariable Laboratory Process with an Adjustable Zero", *IEEE Transactions on Control Systems Technology*, Vol. 8, No. 3 (May), 456-465.

Martin C; A. Urquia; J. Sanchez; S. Dormido; F. Esquembre; J.L. Guzman and M. Berenguel. 2004 "Interactive Simulation of Object-Oriented Hybrid Models, by Combined use of Ejs, Matlab/Simulink and Modelica/Dymola", *In Proc. of the 18th European Simulation Multiconference*, 210-215 (Magdeburg, Germany).

Martin C.; A. Urquia and S. Dormido. 2005a. "Object-oriented modelling of interactive virtual laboratories with Modelica". *In Proc. of the 4th Int. Modelica Conference*, 159-168 (Hamburg, Germany).

Martin C.; A. Urquia and S. Dormido. 2005b. "Object-oriented modeling of virtual laboratories for control education". *In Proc. of the16th IFAC World Congress*, paper code Th-A22-TO/2 (Prague, Czech Republic).