Implementation of Interactive Virtual Laboratories for Control Education using Modelica

Carla Martin, Alfonso Urquia and Sebastian Dormido

Abstract—A novel approach to the implementation of interactive virtual-labs is proposed. The model and the view of the virtual-lab are described in Modelica language, and the virtual-lab is run using Dymola. To achieve this goal, a systematic methodology to transform any Modelica model into a formulation suitable for interactive simulation has been developed. In addition, VirtualLabBuilder Modelica library has been programmed. This library contains a set of ready-to-use Modelica models of visual interactive elements (i.e., containers, animated geometric shapes and interactive controls), that allows easy creation of the virtual-lab view.

This approach has two strong points. Firstly, it allows taking advantage of the Modelica capabilities for multi-domain modeling and for model reuse. In particular, existing Modelica libraries for modeling of physical systems can be reused in order to build the virtual-lab models. Secondly, VirtualLabBuilder library allows performing an object-oriented description of the virtual-lab view, which facilitates its development, maintenance and reuse.

The proposed approach is discussed in this manuscript and it is illustrated by means of a case study: the implementation of the virtual-lab of an industrial boiler, intended for education on chemical-process control.

I. INTRODUCTION

Virtual laboratories supporting interactive simulations are effective pedagogical resources for control education [1]. During the interactive simulation run, students can change the value of some selected inputs, parameters and state variables of the model, 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. Virtual-labs allow the students to play an active role in their learning process and this promotes their motivation to study the subject.

Virtual-labs are composed of a model and a view. The model is the mathematical model representing the relevant behavior of the system under study. The view is the user-to-model interface. It is intended to provide a visual representation of the simulated model behavior and to facilitate the user’s interactive actions on the model during the simulation run. 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 [2] is a freely available, object-oriented modeling language that facilitates the physical modeling paradigm [3]. Models are mathematically described by differential, algebraic and discrete equations. Modelica supports a declarative (i.e., non-causal) description of the model. Therefore, the use of Modelica reduces considerably the modeling effort and permits better reuse of the models.

Modelica is intended for multi-domain modeling. Currently, a number of free and commercial component libraries in different domains are available [2], including electrical, mechanical, thermo-fluid and physical-chemical. Modelica is well suited for describing the type of multi-domain models used in automatic control.

However, neither Modelica language nor Modelica simulation environments (i.e., Dymola [4], OpenModelica [5], etc.) support interactive simulation. As a consequence, extending Modelica capabilities in order to facilitate interactive simulation (i.e., virtual-lab implementation) is an open research field.

Previous work on this topic includes [6], [7], [8], [9]. The combined use of Modelica/Dymola and other software tools is proposed in [7], [8], [9]. The virtual-lab model is described using Modelica and translated using Dymola. The virtual-lab view is implemented using software tools suited for building interactive user interfaces. In particular, the combined use of Modelica/Dymola, Matlab and Easy Java Simulations [10], [11] is proposed in [7], [8], [9]; and the combined use of Modelica/Dymola and Sysquake [12] is proposed in [8].

A. Contributions of this Paper

An approach to the implementation of virtual-labs using Modelica language is proposed. To achieve this goal, the following two tasks have been completed:

1) A systematic methodology to transform any Modelica model into a formulation suitable for interactive simulation has been proposed. Modelica models adapted according to this methodology can be used to set up interactive virtual-labs.

2) VirtualLabBuilder Modelica library has been designed and programmed. It includes Modelica models implementing graphic interactive elements, such as containers, animated geometric shapes (polygon and oval) and interactive controls (slider and radio-button). These models allow the user to compose the view and to link the view visual properties with the model variables.

The proposed approach to the implementation of virtual-labs is discussed in this manuscript and it is illustrated by...
means of a case study: the set up of a virtual-lab emulating an industrial boiler with two different control strategies.

II. OVERVIEW OF THE PROPOSED APPROACH

The virtual-lab definition includes the description of the model, the view and the bidirectional flow of information between the model and the view.

A. Description of the Virtual-Lab Model

Modelica models can be adapted to suit interactive simulation. The interactive model needs to support instantaneous (i.e., discontinuous) changes in the value of the state variables, parameters (i.e., time-independent properties of the physical system) and user controlled variables (i.e., input variables whose value is interactively set by the user). The values of the parameters and the user controlled variables remain constant between consecutive interactive changes.

Modelica supports instantaneous changes in the value of the model state variables. The state re-initialization is performed using Modelica when clauses and the reinit built-in operator of Modelica.

Interactive parameters and user controlled variables are defined as constant state variables (i.e., with zero time-derivative). Therefore, the changes in the value of these parameters and input variables are implemented by re-initializing their values.

B. Description of the Virtual-Lab View

The Modelica description of the virtual-lab view has to be performed extending and connecting the required graphic components of the VirtualLabBuilder library (see Fig. 1b). The architecture, components and use of VirtualLabBuilder will be described in Sections III and IV. Nevertheless, a brief overview of the view definition procedure is discussed next.

The Modelica class describing the view must be a subclass of PartialView class. This class (i.e., PartialView) is included in VirtualLabBuilder library. It contains the code required to perform the communication between the model and the view. This code is valid for any model and view descriptions, and the virtual-lab designer only needs to set the length of the model-to-view communication interval (i.e., Tcom parameter).

The graphic components have to be connected by the virtual-lab programmer forming a structure. The “root” graphic component (i.e., the container component which hosts the rest of the components), named root, is pre-defined in the PartialView class. The connections among the graphic components determines their layout in the virtual-lab view. Modelica modeling environments (i.e., Dymola [4], OpenModelica [5], etc.) allow defining in a drag-and-drop way the instantiation of the required VirtualLabBuilder library components and connecting them using the mouse.

C. Virtual-Lab Set Up

The Modelica description of the virtual-lab has to be an instance of VirtualLab class. This Modelica class is included in VirtualLabBuilder library. It contains two parametrized generic classes [13]: the classes of the virtual-lab model and view. In addition, the virtual-lab designer has to specify, writing the required equations, how the variables of the model and the view class are connected.

D. Translation to Executable Code and Launch

The virtual-lab description, obtained as discussed in Section II-C, is translated using Dymola and ran. 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 virtual-lab 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 packed 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 [4] is the server and the Java program (which is automatically generated during the model initialization) is the client.

Once the jar file has been created, it has to be executed by the virtual-lab 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.

## III. ARCHITECTURE OF VirtualLabBuilder LIBRARY

VirtualLabBuilder library is composed of the following five packages (see Fig. 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 to define some properties of the graphic interactive elements (such as color, layout, etc.).

## IV. INTERACTIVE GRAPHIC ELEMENTS

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

- **Containers** (MainFrame, Dialog, Panel and DrawingPanel classes). These graphic elements that 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, Oval, Text and Arrow classes). These graphic 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, Label 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.

### A. Containers

MainFrame class creates a window where containers and interactive controls can be placed. The view can contain only one MainFrame object. The user can stop the simulation by closing this window. This class has the following parameters:

- position: location of the window on the computer screen in pixels.
- 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 and FlowLayout.

Dialog class, like MainFrame, creates a window where containers and interactive controls can be placed. This class has only two differences with MainFrame class: simulation run doesn’t stop by closing this window and there can be more than one Dialog object.

Panel class creates a panel where containers and interactive controls can be placed. This class has the following parameters:

- position: location of the panel within the container.
- layoutPolicy: layout policy of the element. It sets where the elements placed within the panel are located. Possible values are BorderLayout, GridLayout and FlowLayout.

DrawingPanel class creates a two-dimensional container that only can contain drawable objects (i.e., Polygon, Oval, Text and Arrow 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.

### B. Drawables

Oval class draws an oval. The 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.
- intLineColor: indicates whether the line color changes during the simulation or remains constant.
- intFillColor: indicates whether the filling color 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.
• intVerticesX[]: array that indicates whether the horizontal position of each polygon’s point changes during the simulation or remains constant.
• intVerticesY[]: array that indicates whether the vertical position of each polygon’s point changes during the simulation or remains constant.
• intLineColor: indicates whether the line color changes during the simulation or remains constant.
• intFillColor: indicates whether the filling color changes during the simulation or remains constant.

Text class displays a string. The position of the string center can be linked to the model variables. The class has the following parameters:

• textColor: color of the string.
• intCenter: indicates whether the center of the string change during the simulation or remain constant.
• textString: the string with the text to draw.

Arrow class displays a vector. The position of the origin and vertical and horizontal components of the vector can be linked to the model variables. The class has the following parameters:

• color: color of the arrow.
• intOrigin: indicates whether the vector origin change during the simulation or remain constant.
• intLength: indicates whether the horizontal and vertical components of the vector change during the simulation or remain constant.

C. 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. This class has the following parameters:

• position: radio button position inside the container object.
• buttonValue: initial value of the button variable (true or false).
• buttonGroup: name of the group that the radio button belongs to.

Label class creates a decorative label. This class has the following parameters:

• position: label position inside the container object.
• text: text to be displayed.
• alignment: indicates how to align the text of the element.
• backgroundColor: color used for the background of the object.
• foregroundColor: color used for the foreground of the object.
• typeofFont: font used for the text displayed by the object.
• styleofFont: font style used for the text displayed by the object.

D. Connection Rules

The interface of the interactive graphic components is composed of connectors, which facilitate the connection among the components. Four connector types have been defined. Each one has a distinctive icon. Connector icons are squared or circular, empty or filled.

Two types of interfaces have been defined (see Fig. 1.b):

1) The interface of the container components (i.e., MainFrame, Dialog, Panel and DrawingPanel) has three connectors: two placed on one side (called “left connectors”) and the third one (called “right connector”) placed on the opposite side.

2) The interface of the interactive controls (i.e., Slider, RadioButton and Label) and the drawable elements (i.e., Polygon, Oval, Text and Arrow) has two connectors (called “left connectors”: one filled and one empty).

The virtual-lab programmer must observe the following three rules when connecting the graphic elements:

1) Only connectors with the same shape (circular or squared) can be connected.
2) Each filled connector must be connected to one and only one empty connector.
3) Each empty connector can be left unconnected or can be connected to one and only one filled connector.

The meaning of the connections among the graphic components is as follows:

1) When two components are connected using their “left connectors”, then both components are hosted within the same container. The component position in the chain of connected elements determines its insertion order within the container.

2) When two components are connected using the “right connector” of the first component and a “left connector” of the second component, then the second component is hosted within the first component.

An example is shown in Fig. 2. The mainFrame and dialog components are hosted in root. The drawingPanel and Ti1 components are hosted in mainFrame. The line1 and line2

![Diagram](image.png)

Fig. 2. Example of connection among graphic components
components are hosted in drawingPanel. Finally, the Kp1 component is hosted in dialog.

V. CASE STUDY

The implementation with Modelica of a virtual-lab is discussed. The virtual-lab is used for education on automatic control. It is intended to illustrate the dynamic behavior of an industrial boiler operating under two different control strategies: manual and decentralized PID.

A. Description of the Virtual-Lab Model

The mathematical model of the boiler has been extracted from [14]. The input of liquid water is placed at the boiler bottom, and the vapor output valve is placed at the top. The water contained in the boiler is continually heated.

The boiler model has been composed using components from JARA 2i [15], a version of JARA library [16], [17] that has been adapted for interactive simulation by applying the methodology proposed in [8], [9]. JARA is a Modelica library of some fundamental physical-chemical principles. JARA's main application domain is the modeling of transport, thermo-fluid, phase change and chemical processes in the context of automatic control.

The model diagram of the boiler and its control system is shown in Fig. 3. Two control volumes have been considered: (1) a control volume containing the liquid water stored in the boiler; and (2) a gaseous control volume containing the vapor. The vapor volume is equal to the difference between the boiler-recipient inner volume and the water volume. The boiling is a transport phenomena represented by a model connecting both control volumes. The heat-flow into de boiler, the pressure at the valve output and the water pump are modeled using JARA source models.

B. Composing the Virtual-Lab View

The Modelica description of the virtual-lab view, shown in Fig. 4, generates the interactive graphic interface shown in Fig. 5. The relationship between the Modelica description and the corresponding graphic interface is briefly explained next.

The Modelica model describing the view must extend the PartialView class, which contains one pre-defined graphic element: root.

The root component has two components hosted in it: mainFrame and dialog. These components are objects of MainFrame and Dialog class respectively. They generate the two windows shown in Fig. 5.

The mainFrame layout policy is set to BorderLayout, in order to allow selecting the position of the hosted elements (i.e., north, south, center, east or west positions). In this case, three containers are placed inside mainFrame: drawingPanel (of DrawingPanel class), and panelNorth and panelSouth (of Panel class).

- drawingPanel is placed in the center of the mainFrame. This component contains the animated diagram of the plant. This diagram is composed of drawable elements of Polygon, Oval, Text and Arrow classes. The liquid, heating system, pump and valve are represented by components of Polygon class. The two controllers are represented by components of Oval class and the set-point of the liquid volume is represented by a component of Arrow class.
- panelNorth hosts interactive controls of RadioButton and Slider classes. The two radio-buttons allow the user to select the control strategy (manual or decentralized PID). The two sliders allow the user to change the pump input flow and the heater heat-flow when the manual control strategy is selected.
- panelSouth hosts interactive controls of Label and Slider classes. These sliders allow the user to perform interactive changes in the value of the boiler volume, the output pressure, the valve opening, the water volume and the vapor flow set points, the mass and temperature of the water, and the vapor moles contained inside the boiler.

The dialog container hosts interactive controls of the Slider class. These sliders allow the user to change the parameter values of the two PID controllers.

C. Virtual-Lab Set Up and Launch

The virtual-lab description is obtained as discussed in Section II-C. It is translated using Dymola and ran. Then, the jar file containing the Java code of the virtual-lab view is automatically generated. When the virtual-lab user executes this jar file, the virtual-lab view is displayed (see Fig. 5).

The time evolution of some selected variables can be plotted using Dymola. The dynamic response of the boiler to a step change in the output pressure is shown in Fig. 6. This
Fig. 4. Diagram of the Modelica description of the view

Fig. 5. Virtual-lab view
change has been interactively performed by the virtual-lab user at the simulated time 170 s. The boiler is operating in automatic control mode. The following four plots are shown in Fig. 6:

- The actual value of the vapor flow and its setpoint are shown in Fig. 6a.
- The heat generated by the heater is shown in Fig. 6b.
- The actual value of the water volume contained inside the boiler, and its setpoint value, are shown in Fig. 6c.
- The liquid flow rate generated by the pump is displayed in Fig. 6d.

VI. CONCLUSIONS

A novel approach to the virtual-lab implementation using Modelica language has been proposed and it has been successfully applied. 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.

Some advantages of this approach are the following. Firstly, the virtual-lab is completely described using Modelica language, an object-oriented modeling language aimed to be a de-facto standard for representing models and to support model exchange. Secondly, existing Modelica libraries for modeling of physical systems can be reused in order to build the virtual-lab models. Finally, the virtual-lab view is object-oriented modeled, which facilitates its development, maintenance and reuse.

VII. ACKNOWLEDGMENTS

This work has been supported by the Spanish CICYT, under DPI2004-01804 grant, and by the IV PRICIT (Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid, 2005-2008), under S-0505/DPI/0391 grant.

REFERENCES