Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

C. Martin,A. Urquia, S. Dormido
UNED Madrid, Spain

Modeling of Interactive Virtual Laboratories with Modelica

pp. 159-168

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Modeling of Interactive Virtual Laboratories with Modelica

Carla Martin Alfonso Urquia Sebastian Dormido
Departamento de Informatica y Automatica, E.T.S. de Ingenieria Informatica, UNED
Juan del Rosal 16, 28040 Madrid, Spain

Abstract

The implementation of virtual-labs supporting runtime and batch interactivity is discussed and it is illustrated by means of several case studies. The virtual-lab *models* have been programmed using Modelica language and translated using Dymola. The virtual-lab *views* (i.e., the user-to-model interfaces) have been implemented using Ejs and Sysquake. This software combination approach allows us to take advantage of the best features of each tool. Ejs and Sysquake capability for building interactive user interfaces composed of graphical elements, whose properties are linked to the model variables. Modelica capability for physical modeling and Dymola capability for simulating hybrid-DAE models.
In order to implement this approach, the following tasks have been completed: (1) a novel modeling methodology, adequate for runtime interactive simulation using Ejs, Simulink and Modelica/Dymola, has been proposed; and (2) a Sysquake to Dymosim interface has been programmed: a set of functions in LME, intended to be used by the Sysquake applications.

1 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 web-based and distance education [1].
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 bidirectional 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.
Two alternative types of interactivity can be implemented:

- *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.

- *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.

1.1 Contributions of this paper

The implementation of interactive virtual-labs is discussed in this manuscript. Runtime and batch interactivity are considered. In both cases, the *models* are programmed using Modelica language and translated using Dymola [2]. The *view* of the virtual-labs supporting *runtime interactivity* has been implemented using Easy Java Simulations [3] (abbreviated: Ejs. http://fem.um.es/Ejs/). The *view* of the virtual-labs supporting *batch interactivity* has been programmed using Sysquake (http://www.calerga.com/).
This software combination approach allow us to take advantage of the best features of each tool. Ejs and Sysquake capability for building interactive user-interfaces composed of graphical elements, whose properties are linked to the model variables. Modelica capability for physical modeling, and finally Dymola capability for simulating hybrid-DAE models.

The tasks completed to successfully implement this approach are discussed. In particular:

– **Runtime interactive simulation.** The communication between the virtual-lab view (programmed using Ejs) and the virtual-lab model (C-code generated by Dymola) is accomplished by using the Ejs-Simulink and the Dymola-Simulink interfaces. The C-code generated by Dymola for the Modelica model can be embedded within a Simulink block [2]. On the other hand, Ejs allows the model to be partially or completely developed using Simulink block diagrams. As a consequence, virtual-labs supporting runtime interactivity can be implemented by combining the use of Ejs, Matlab/Simulink and Modelica/Dymola.

  The Modelica model needs to be adequately formulated in order to be: (1) useful as a Simulink block; (2) able to accept information from the virtual-lab view; and (3) able to return information to the virtual-lab view. As a consequence, a modeling methodology has been proposed. It states how a Modelica model can be formulated to suit *runtime interactive simulation*. This methodology has been successfully applied to program a set of virtual-labs for chemical process control. One of them is discussed in this manuscript: the virtual-lab of a double-pipe heat exchanger. Other virtual-labs are discussed in [4, 5, 6].

– **Batch interactive simulation.** A set of Sysquake functions has been programmed to facilitate data exchange between the view and the model of the virtual-lab. These functions synchronizes the execution of the *dymosim.exe* file (generated by Dymola) and the Sysquake application. The combined use of Sysquake and Modelica/Dymola for virtual-lab programming is illustrated by means of two case studies.

2 Runtime interactive simulation, by combining the use of Ejs, Simulink and Modelica/Dymola

Easy Java Simulations (Ejs) is a open source, Java-based software tool intended to implement virtual-labs. It can be freely downloaded from the web-site http://fem.um.es/Ejs/. Ejs guides the user in the process of creating the *model* and the *view*, generates the Java source code of the virtual-lab program, compiles the program, packs the resulting object files into a compressed file, and generates HTML pages containing the virtual-lab as an applet. Then, the user can readily run the virtual-lab and/or publish it on the Internet.

The *view* definition is a strong point of Ejs. Ejs includes a set of ready-to-use visual elements, that the modeller can use to compose a sophisticated view in a simple, drag-and-drop way. The properties of the view elements can be linked to the model variables.

On the contrary, the *model* definition and simulation is a weak point of Ejs. Ejs provides its own procedure to define the model, which must be formulated by the user as a sorted sequence of algorithm clauses (i.e., assignment statements). Ejs implements some standard ODE solvers. However, it implements neither algorithms for symbolic formula manipulation nor algebraic-loop solvers.

Ejs version 3.3 (release 2004) provides a Ejs to Matlab/Simulink interface. Therefore, Ejs 3.3 supports the option of describing and simulating the model using Matlab/Simulink: (1) Matlab code and calls to any Matlab function can be used at any point in the Ejs model; and (2) the Ejs model can be partially or completely developed using Simulink block diagrams. This significantly improves the Ejs capabilities for model description and numerical solution. However, Simulink modeling paradigm (i.e., graphical block-diagram modeling) exhibits some limitations [7]. It requires explicit state models (ODE) and that the blocks have a unidirectional data flow from inputs to outputs. These restrictions strongly condition the modeling task, which requires a considerable effort from the modeller.

The use of Modelica language is an attractive alternative to Simulink, because it reduces considerably the modeling effort and permits better reuse of the models. The combined application of Modelica/Dymola and Ejs to the implementation of virtual-labs in discussed next.

2.1 Combined use of Ejs, Matlab/Simulink and Modelica/Dymola

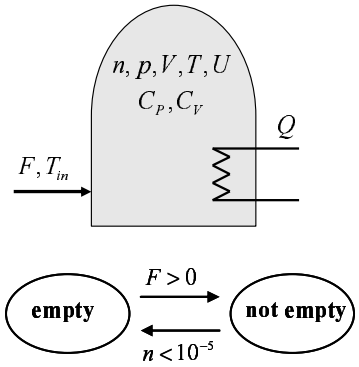
Dymola 5.0 interface to Simulink 3.0 can be found in Simulink's library browser: DymolaBlock block [2]. This block is an interface to the C-code generated by Dymola for the Modelica code. DymolaBlock block can be connected to other Simulink blocks, and also to other DymolaBlocks blocks, in the Simulink's workspace window. Simulink synchronizes the nu-

merical solution of the complete model, performing the numerical integration of the DymolaBlock blocks together with the other blocks.
In order to make the Modelica model useful as a DymolaBlock block, the computational causality of the Modelica model interface needs to be explicitly set [2]. The input variables are supposed to be calculated from other Simulink blocks, while the output variables are calculated from the Modelica model.
Ejs 3.3 supports the option of describing and simulating the virtual-lab model using Simulink. In this case, the data exchange between the virtual-lab view (composed using Ejs) and the model (Simulink block diagram) is accomplished through the Matlab workspace. The properties of the Ejs' view elements are linked to variables of the Matlab workspace, which can be written and read from the Simulink block diagram.
The Modelica model needs to be built to allow the communication with the virtual-lab view. It needs to support the discontinuous changes in the value of its state variables, parameters and input variables which are the result of the user interaction. In some cases, several choices of the state variables need to be supported simultaneously in the model, in order to provide the user with alternative ways of describing the state changes. A design methodology for the Modelica model is described in Section 2.2. Further details can be found in [4, 6].

2.2 Modeling methodology

The model of a perfect gas is shown in Figure 1. The input flow of gas (F), of heat (Q) and the input temperature (T_{in}) are input variables. The gas volume (V) and the heat capacities (C_P, C_V) are time-independent properties of the physical system.
In general, different choices of the model state-variables are possible. Possible choices in the model shown in Figure 1 include: $e_1 = \{p, T\}$, $e_2 = \{n, T\}$ and $e_3 = \{n, p\}$; where e_i represents one particular choice of the state variables. If the user wants to change interactively p and T, the appropriate choice is $e_1 = \{p, T\}$. This is also the right choice if the user wants to change p and to keep constant T, or of he wants to change T and to keep constant p. Likewise, the appropriate choice is e_2 if the user wants: (1) to modify interactively n and T; or (2) to modify n and to maintain constant T; or (3) to modify T and to maintain constant n. An analogous reasoning is applied to e_3. In general, an interactive model is required to support state changes that correspond with different choices of the state variables.
In addition, interactive changes of the model parameters can have different effects depending on the state variable choice. Consider an instantaneous change in the gas volume (V) of the model shown in Figure 1. If the state variables are $e_1 = \{p, T\}$, then the change in V produces an instantaneous change in the number of moles (n), while the pressure (p) and the temperature (T) remain constant. On the contrary, if the state variables are $e_2 = \{n, T\}$, then the change of volume produces a change of pressure. In this case, the number of moles (n) and the temperature remain constant. As a consequence, the interactive model needs to support different choices of the state variables simultaneously.
An approach to implement this capability is the following. Building the interactive model as composed of several instantiations of the physical model, each one with a different choice of the state variables. When describing an interactive action on the model, the user selects the adequate state-variable choice according to his preference. This information is transmitted from the virtual-lab *view* to the *model*. Then, the interactive model uses the adequate physical-model instantiation (that with the chosen state selection) for executing the instantaneous change in the parameters and state variables, and for solving the re-start problem. Finally, these calculated values are used to re-initialize the other physical-model instantiations. This action guarantees that all physical-model instantiations describe the same trajectory.
Modelica capability for state-selection control allows easy implementation of this approach [8]. Three instantiations of the perfect-gas model (i.e., *perfectGas*) have been defined (see Figure 2): (1) *perfectGasSS1*, with $e = \{p, T\}$; (2) *perfectGasSS2*, with $e = \{n, T\}$; and (3) *perfectGasSS3*, with $e = \{n, p\}$. The Appendix A provides the Modelica code for the perfect-gas model.
Two input variables to the DymolaBlock block are used to carry out the interactive changes in the state: *Istate*[:] and *CKstate*[:] (see Figure 2).
The array *Istate*[:] contains the values used to re-initialize the model state. In the perfect-gas model: $Istate[:] = \{n, p, T\}$.
The array *CKstate*[:] is used to trigger the state re-initialization events, which are performed using the Modelica operator *reinit*. Each variable of the array *CKstate*[:] is used to trigger the events in a different instantiation of the physical model. The perfect-gas model contains three instantiations of the physical-model: *perfectGasSS1*, *perfectGasSS2* and *perfectGasSS3*. Consequently, the array *CKstate*[:] has three

n: Number of moles
p: Absolute pressure
V: Volume of the gas
T: Absolute temperature
U: Internal energy
C_P, C_V: Heat capacities
F: Input flow of gas
T_{in}: Input temperature
Q: Input flow of heat
R: Perfect gas constant



$$p \cdot V = n \cdot R \cdot T$$

$$\frac{dn}{dt} = \begin{cases} 0 & \text{empty} \\ F & \text{not empty} \end{cases}$$

$$\frac{dU}{dt} = \begin{cases} 0 & \text{empty} \\ F \cdot C_P \cdot T_{in} + Q & \text{not empty and } F > 0 \\ F \cdot C_P \cdot T + Q & \text{not empty and } F \leq 0 \end{cases}$$

$$U = n \cdot C_V \cdot T$$

$$C_P - C_V = R$$

Figure 1: Model of a perfect gas

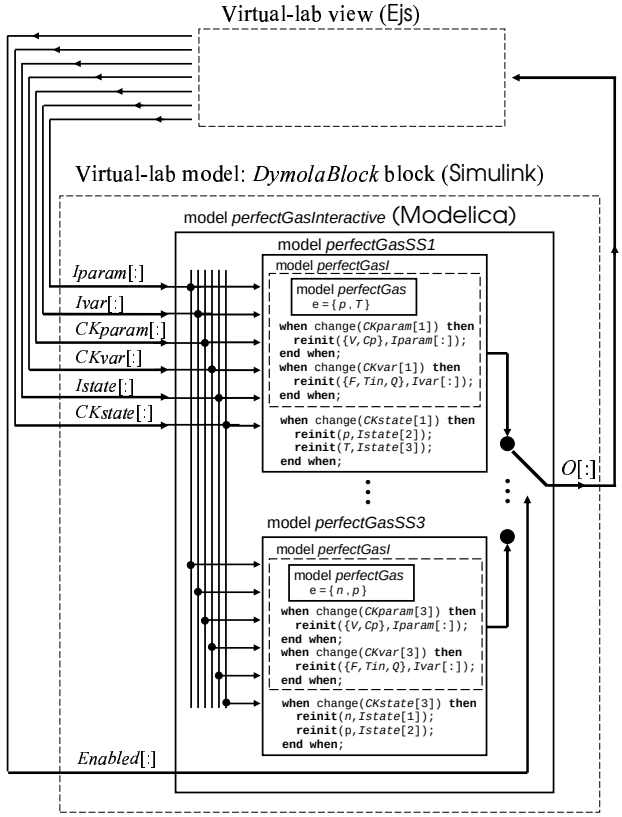


Figure 2: Schematic description of the perfect-gas virtual-lab

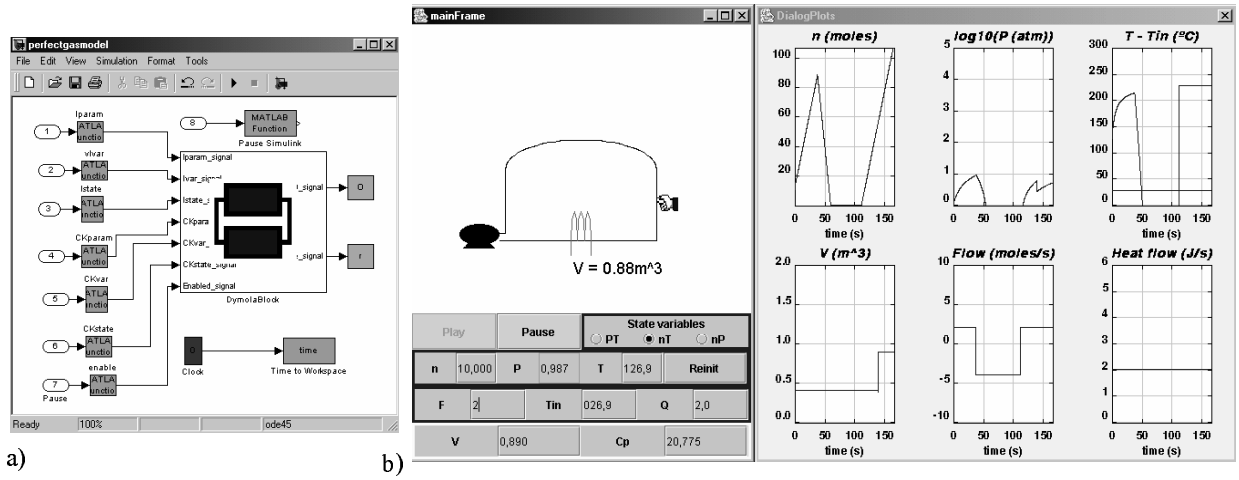
components. *CKstate*[1] triggers the change in the state-variables of *perfectGasSS1*. *CKstate*[2] and *CKstate*[3] trigger the change in the state-variables of *perfectGasSS2* and *perfectGasSS3* respectively (see Figure 2).

The interactive parameters (V, C_P) and the input variables (F, T_{in},Q) are defined as constant state-variables (i.e., with zero time-derivative) in the physical model [4]. Their values are changed by using the *reinit* operator. Four input variables to the DymolaBlock block are used (see Figure 2): two arrays (*Iparam*[:], *Ivar*[:]) containing the new values, and two arrays (*CKparam*[:], *CKvar*[:]) for triggering the re-initialization events.

The output-variable array of the DymolaBlock block, *O*[:] (see Figure 2), contains the variables linked to the properties of the virtual-lab *view*. Ejs uses the value of this output array (*O*[:]) to refresh the simulation view. The value of the input array *Enabled*[:] is set by Ejs, and it selects which output is connected to the output signal *O*[:]. The output array in the perfect-gas model is the following: $O[:] = \{n, p, T, V, C_P, T_{in}, F, Q\}$.

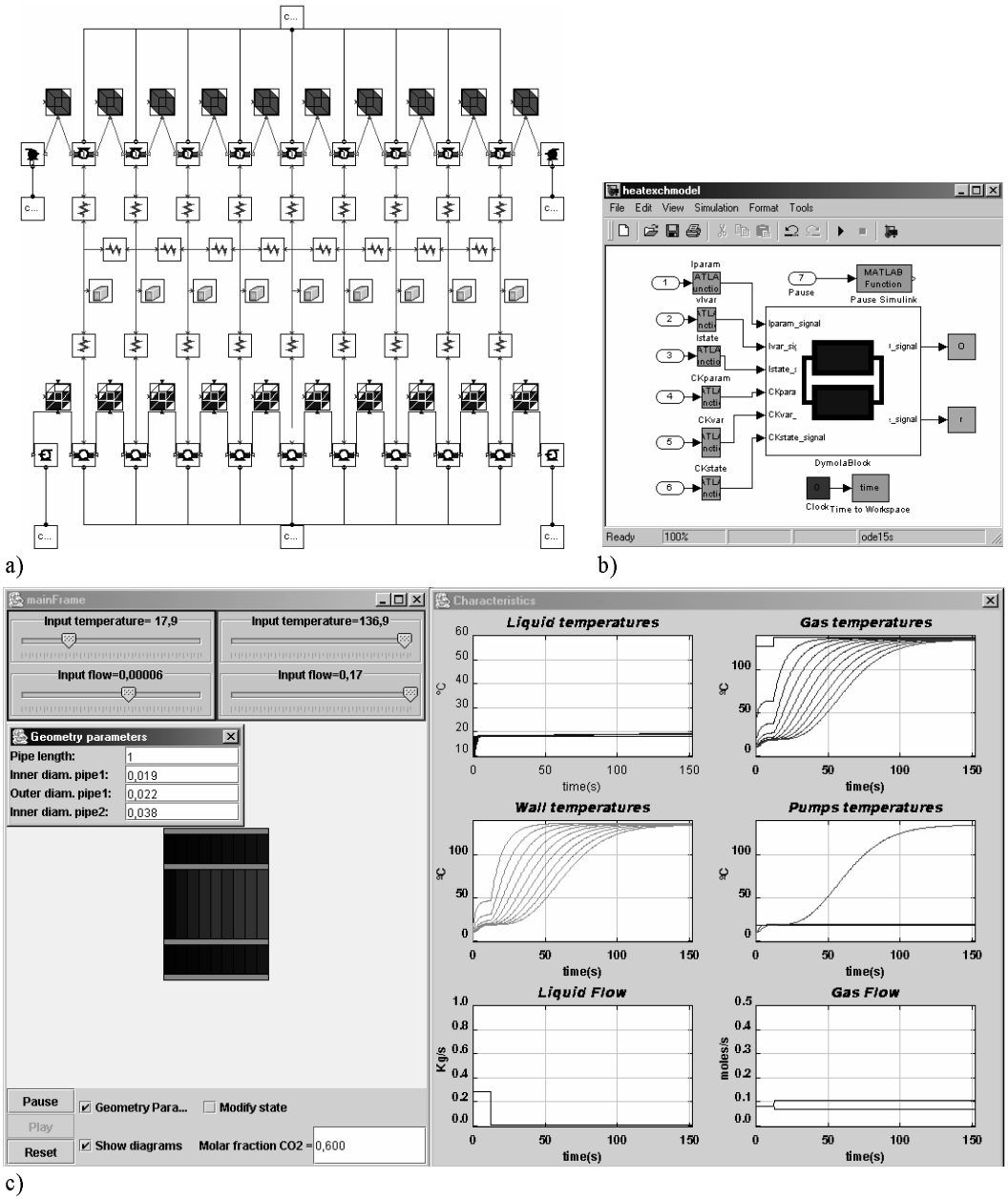
The Simulink model of the perfect-gas is shown in Figure 3a. The Modelica model (*perfectGasInteractive*) is embedded within the DymolaBlock block. The blocks connected to the DymolaBlock inputs (*"MATLAB Fcn"* blocks) transmit the value of the input variables from the Matlab workspace to the Simulink block-diagram window. The blocks connected to the DymolaBlock outputs (*"To Workspace"* blocks) transmit the value of the output variables from the Simulink block-diagram window to the Matlab workspace. Ejs reads the value of these output variables from the Matlab workspace and writes the value of the input variables in the Matlab workspace.

The view of the virtual-lab is shown in Figure 3b. The main window (on the left side) contains the schematic diagram of the process (above) and the control buttons (below). Both of them allow the user to experiment with the model. The vessel volume, represented in the schematic diagram, is linked to the V variable. Its value can be interactively changed by clicking on the hand picture and dragging the mouse. Three radio buttons allow choosing the state variables ($\{p,T\}$, $\{n,T\}$ or $\{n,p\}$). Text fields allow the user set the value of the state variables (n, p, T), the input variables (F, T_{in}, Q) and the parameters (V, C_P). The window placed on the right side of the virtual-lab view contains graphic plots of the model variables.



a) b)

Figure 3: Perfect-gas virtual-lab: a) Simulink model; b) View



a) b)

c)

Figure 4: Heat exchanger virtual-lab: a) Physical model; b) Simulink model; c) View

2.3 Case study I: heat exchanger

The interactive simulation of a heat exchanger has been implemented, by the combined use of Ejs, Matlab/Simulink and Modelica/Dymola. A mixture of carbon dioxide and sulfur dioxide is cooled by water in a double-pipe heat exchanger [9]. Two modes of operation are allowed: cocurrent or parallel flow and countercurrent flow. The convective heat transfer on both the tube and shell sides are calculated from the Dittus-Boelter correlation [9]. The center heat exchanger tube is made of copper with a constant thermal conductivity, and the exterior of the steel pipe shell is very well insulated.

The physical model of the heat exchanger has been composed using JARA. The model diagram is shown in Figure 4a. JARA is a set of libraries of some fundamental physical-chemical principles. JARA was originally written in Dymola language [10, 11]. Later on, it was translated into Modelica language. The methodology discussed in Section 2.2 was applied in order to make JARA useful for interactive simulation [5].

JARA is composed of seven model libraries, including models of:

- *Control volumes* containing: (1) an ideal mixture of an arbitrary number of semi-perfect gases; or (2) a homogeneous liquid mixture composed of an arbitrary number of components; or a homogeneous solid. The liquid and gaseous control volumes are considered open systems (i.e., they can exchange mass and heat with their environment) and chemical reactions can take place inside them. The solid control volumes are considered closed systems (i.e., they only exchanges energy, not mass, with their environment).

- *Mass transport* due to the pressure and concentration gradient, the gravitational acceleration, chemical reactions, liquid-vapor phase changes, etc.

- *Heat transport* by conduction and convection.

The Simulink model is shown in Figure 4b. The interactive model of the heat exchanger, written in Modelica language, has been embedded within the DymolaBlock block. Observe that the structure of this Simulink model is completely analogous to the perfect-gas model, shown in Figure 3a.

The view of the virtual-lab is shown in Figure 4c. The main window (on the left side) contains: (1) a diagram of the heat exchanger; (2) buttons to control the simulation run (i.e., pause, reset and play); (3) sliders and a text field to modify the input variables (i.e., liquid and gas flows, liquid and gas input temperatures, and molar fraction of CO_2 and SO_2 in the gas mixture); and (4) checkboxes to show and hide three secondary windows: *"Geometry Parameters"*, *"Modify State"* and *"Characteristics"*.

The *"Geometry Parameters"* window contains text fields that can be used to modify the pipe length and diameters. The controls placed in the *"Modify State"* window allow changing the temperature of the medium inside each control volume (i.e., the cooling liquid, the gas mixture or the metal wall). Finally, "Characteristics" is a window with several plots of the model variables.

3 Batch interactive simulation, by combining the use of Sysquake and Modelica/Dymola

Sysquake is a commercial tool intended to develop interactive applications [12]. It is based on LME, an interpreter specialized for numerical computation. LME is mostly compatible with the language of MATLAB(R) 4.x and it includes many features of MATLAB 5 to 7. It implements graphic functions specific to dynamic systems (such as step responses and frequency responses) and general purpose functions used for displaying any kind of data.

Typically, a Sysquake application contains several interactive graphics, which are displayed simultaneously. These graphics contain elements that can be manipulated using the mouse. While one of these elements is being manipulated, the other graphics are automatically updated to reflect this change. The content represented by each graphic, and its dependence with respect to the content of the other graphics, is programmed using LME.

The main goal of Sysquake is the interactive manipulation of graphics. The user can define functions, called *handlers*, intended to perform different tasks managed by Sysquake. These tasks include the model initialization, manipulation of figures and selection of menus.

As input and output, the *handlers* use variables as well as values managed directly by Sysquake, such as the position of the mouse. Therefore, only the code necessary for displaying the figures and processing manipulations from the user is required. This results in small scripts, developed quickly and easy to maintain.

LME can be extended by libraries, composed of related functions written in LME, or by extensions de-

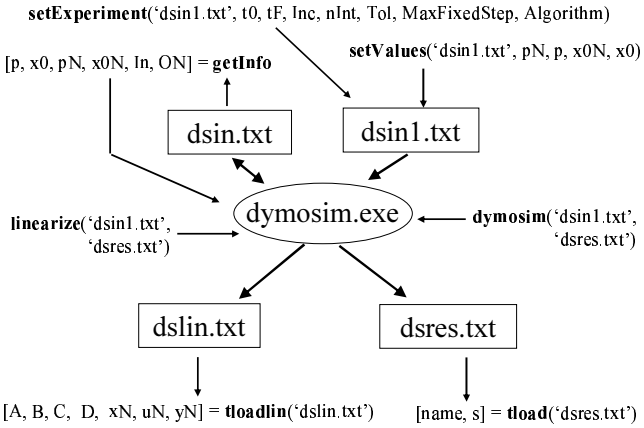


Figure 5: Sysquake-Dymosim interface functions

veloped with standard compilers.

3.1 Combined use of Sysquake and Modelica/Dymola

A Sysquake interface to Dymosim (i.e., the executable file generated by Dymola [2]) has been programmed. This interface is a set of functions in LME, intended to be used by the Sysquake applications. These functions perform the following tasks:

- The *setExperiment* and *setValues* functions write the experiment description to a text file. This text file is intended to be the input file for *dymosim.exe*.
- The *dymosim* and *linearize* functions execute the *dymosim.exe* file in order to simulate and linearize the Modelica model respectively.
- The *tload* and *tloadlin* functions: (1) read the output file generated by *dymosim.exe* after a model simulation or linearization respectively; and (2) save these results as variables to the Sysquake workspace. These variables can be used by Sysquake applications.

Next, a brief description of each function is provided (see Figure 5):

- *setExperiment(txtFile, StartTime, StopTime, Increment, nInterval, Tolerance, MaxFixedStep, Algorithm)*. It writes to the *txtFile* text file (default file name: *dsin1.txt*) the simulation parameters.
- *[p, x0, pN, x0N, InputN, outputN] = getinfo*. This function executes the *dymosim.exe* file (command *dymosim -i*) in order to generate the Dymosim input file (*dsin.txt*). In addition, this function reads the names of the model variables (i.e., inputs, outputs, parameters, states) and their default values from *dsin.txt* file, and saves them as variables to the Sysquake workspace.
- *SetValues(txtFile, pN, p, x0N, x0)*. The name and the value of the model parameters and state variables are written to the *txtFile* text file (*dsin1.txt* by default).
- *dymosim(iFile, oFile)*. This function executes the following command: *dymosim -d dsin.txt iFile oFile*. The default file name for *iFile* and *oFile* is *dsin1.txt* and *dsres.txt* respectively.
- *linearize(iFile, oFile)*. This function obtains the linearized model by executing the command: *dymosim -l iFile oFile*. The default file name for *iFile* and *oFile* is *dsin1.txt* and *dslin.txt* respectively.
- *[N,s] = tload(oFile)*. This function reads the result file, *oFile* (default file name: *dsres.txt*), and stores the signal names and the simulation results into *N* (text matrix) and *s* (numeric matrix) respectively.
- *[A,B,C,D,xN,uN,yN] = tloadlin(txtfile)*. It loads the linear model generated by dymosim from the *txtfile* result file (default file name: *dslin.txt*) into the Sysquake workspace.

Next, two case studies are provided to illustrate the use of this Sysquake-Dymosim interface.

3.2 Case study II: control loop

The interactive simulation of the control loop shown in Figure 6 is implemented by combining the use of Sysquake and Modelica/Dymola. The constitutive relation of the hysteresis-based controller in shown in Figure 7. The setpoint is the composition of two signals: a piecewise linear function and a sine function. The model of the control loop has been programmed using Modelica language and translated using Dymola. The execution of the *dymosim.exe* file generated by Dymola is controlled by the Sysquake application (i.e., the virtual-lab *view*).
The *view* of the virtual-lab is the Sysquake application shown in Figure 8. It is composed of four graphics. Three of them are interactive:

- *"Constitutive relation"* plot (graphic on the upper left). The position of the $\{a, b, c, d, e, f\}$ points of

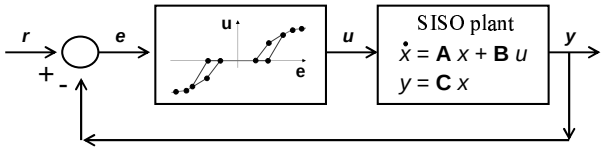


Figure 6: Control loop

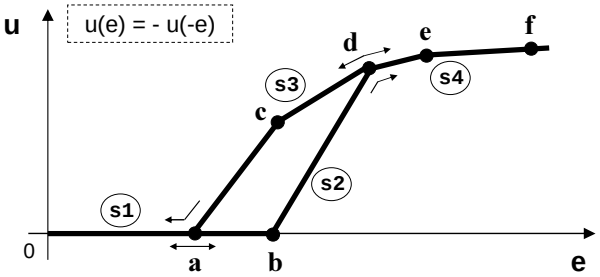


Figure 7: Constitutive relation of the controller

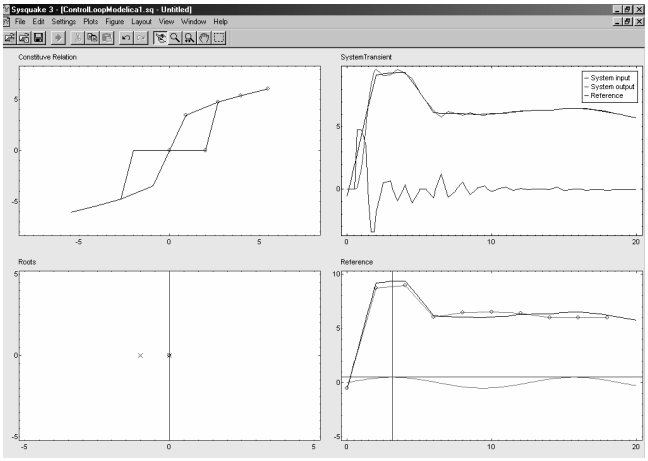


Figure 8: View of the control loop virtual-lab

the controller constitutive relation can be changed by dragging the mouse.

– *"Roots"* plot (graphic on the lower left). The plant zeros and poles can be changed by clicking on the circles and crosses and dragging the mouse.

– *"Reference"* plot (graphic on the lower right). The shape of the piecewise linear function and the amplitude and frequency of the sine function can be modified by clicking on the lines and circles that appear in the graphic and dragging the mouse.

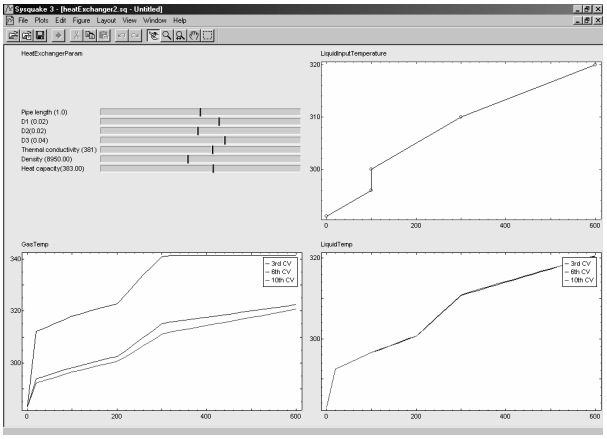


Figure 9: View of the heat exchanger virtual-lab

3.3 Case study III: heat exchanger

The heat exchanger virtual-lab described in Section 2.3 supports runtime interactivity. It was implemented using Ejs, Simulink and Modelica/Dymola. In this section, the heat exchanger model is revisited, and a virtual-lab supporting batch interactivity is programmed by combining the use of Sysquake and Modelica/Dymola.

The *view* of the virtual-lab is the Sysquake application shown in Figure 9. The sliders placed on the upper left side allow modifying some model parameters: the pipe length and diameters, and the thermal parameters of the center heat-exchanger tube.

The graphic on the upper right corner is interactive. It represents the time-evolution of the inlet temperature of the water. The shape of this curve can be changed by clicking on one of the points and dragging the mouse.

The graphics on the lower side of Figure 9 show the time-evolution of the temperature at certain positions of the tube and the shell.

4 Conclusions

The feasibility of combining Modelica/Dymola with Ejs and Sysquake, for implementing runtime and batch interactive simulations respectively, has been demonstrated. Ejs and Sysquake are software tools intended to develop interactive applications. Their strong point is the programming of the virtual-lab *view*. Working together with Modelica/Dymola significantly improves the Ejs and Sysquake capabilities for *model* description and simulation. The use of Modelica language reduces considerably the modeling effort.

In order to implement this software combination ap-

proach, a modeling methodology has been proposed and a Sysquake-Dymosim interface has been programmed. Several case studies of virtual-labs supporting runtime and batch interactivity have been discussed.

Acknowledgements

The authors wish to thank Dr. Yves Piguet (Calerga Sarl, Lausanne, CH) for his constructive comments. This work has been supported by the Spanish CICYT, under DPI2001-1012 and DPI2004-1804 grants.

References

[1] Dormido S. Control learning: Present and Future. In: Annual Reviews in Control, vol. 28, pp. 115–136, 2004.

[2] Dynasim AB. Dymola. User's Manual. Version 5.0a. Dynasim AB. Lund, Sweden.

[3] Esquembre F. Easy Java Simulations: a Software Tool to Create Scientific Simulations in Java. In: Computer Physics Communications, vol. 156, pp. 199–204, 2004.

[4] Martin C, Urquia A, Sanchez J, Dormido S, Esquembre F, Guzman J.L, Berenguel M. Interactive Simulation of Object-Oriented Hybrid Models, by Combined use of Ejs, Matlab/Simulink and Modelica/Dymola. In: Proc. 18th European Simulation Multiconference, pp. 210–215, 2004.

[5] Martin C, Urquia A, Dormido S. JARA 2i - A Modelica Library for Interactive Simulation of Physical-Chemical Processes. In: Proc. European Simulation and Modelling Conference, pp. 128–132, 2004.

[6] Martin C, Urquia A, Dormido S. Object-Oriented Modeling of Virtual Laboratories for Control Education. 16^{th} IFAC World Congress, Praha, Czech Republic, July 2005. Accepted.

[7] Astrom K.J, Elmqvist H, Mattsson S.E. Evolution of Continuous-Time Modeling and Simulation. In: Proc. of the 12^{th} European Simulation Multiconference, Manchester, UK, 1998.

[8] Otter M, Olsson H. New features in Modelica 2.0. In: Proc. 2^{nd} International Modelica Conference, pp. 7.1–7.12, 2002.

[9] Cutlip M.B, Shacham M. Problem Solving in Chemical Engineering with Numerical Methods. Prentice-Hall, 1999.

[10] Urquia A. Modelado Orientado a Objetos y Simulación de Sistemas Híbridos en el Ámbito del Control de Procesos Químicos. PhD. Thesis. Dept. Informática y Automática, UNED, Madrid, Spain, 2000.

[11] Urquia A, Dormido S. Object-Oriented Design of Reusable Model Libraries of Hybrid Dynamic Systems. Mathematical and Computer Modelling of Dynamical Systems, 9(1), pp. 65–118, 2003.

[12] Calerga Sarl. Sysquake 3. User's Manual. Calerga Sarl. Lausanne, Switzerland.

APPENDIX A: Modelica code for the perfect-gas model

```
model perfectGas
  parameter Boolean nIsState, pIsState, TIsState;
  Real n (unit="mol", start=20,
          stateSelect = if nIsState
                        then StateSelect.always
                        else StateSelect.default);
  Real p (unit="N.m-2", start=1e5,
          stateSelect = if pIsState
                        then StateSelect.always
                        else StateSelect.default);
  Real T   (unit="K", start=300,
            stateSelect = if TIsState
                          then StateSelect.always
                          else StateSelect.default);
  Real V   (unit="m3", start=1);
  Real Cp  (unit="J/(Kg.K)", start=5*R/2);
  Real Cv  (unit="J/(Kg.K)");
  Real F   (unit="mol.s-1");
  Real Tin (unit="K");
  Real Q   (unit="J.s-1");
  parameter Real R (unit="J/(mol.K)") = 8.31;
protected
  Real U (unit="J", stateSelect = StateSelect.never);
  Boolean empty (start=false);
equation
  // Interactive parameters
  der(V)  = 0;
  der(Cp) = 0;
  // Input variables
  der(F)   = 0;
  der(Tin) = 0;
  der(Q)   = 0;
  // State equation
  p * V = n * R * T;
  // Mol balance
  der(n) = if empty then 0 else F;
  // Energy balance
  der(U) = if empty then 0
           else if F>0 then F*Cp*Tin+Q else F*Cp*T+Q;
  // Internal energy
  U = n * Cv * T;
  // Mayer law
  Cp - Cv = R;
  // Empty-vessel condition
```

```
  when F > 0 and     pre(empty) or
       n < 1e-5 and not pre(empty) then
     empty = not pre(empty);
  end when;
end perfectGas;

model perfectGasI
  extends perfectGas;
  Modelica.Blocks.Interfaces.InPort Iparam (n=2);
  Modelica.Blocks.Interfaces.InPort Ivar   (n=3);
  Modelica.Blocks.Interfaces.InPort Istate (n=3);
  Real CKparam;
  Real CKvar;
  Real CKstate;
  Modelica.Blocks.Interfaces.OutPort O     (n=8);
protected
  Boolean CKparamIs0 (start = true, fixed=true);
  Boolean CKvarIs0   (start = true, fixed=true);
  Boolean CKstateIs0 (start = true, fixed=true);
equation
  // Interactive change of the parameters
  when CKparam>0.5 and     pre(CKparamIs0) or
       CKparam<0.5 and not pre(CKparamIs0) then
     CKparamIs0 = CKparam < 0.5;
     reinit(V,   Iparam.signal[1]);
     reinit(Cp,  Iparam.signal[2]);
  end when;
  // Interactive change of the input variables
  when CKvar>0.5 and     pre(CKvarIs0) or
       CKvar<0.5 and not pre(CKvarIs0) then
    CKvarIs0 = CKvar < 0.5;
    reinit(F,    Ivar.signal[1]);
    reinit(Tin,  Ivar.signal[2]);
    reinit(Q,    Ivar.signal[3]);
  end when;
   // Output signal
  O.signal = { n, p, T, V, Cp, Tin, F, Q };
end perfectGasI;

model perfectGasSS1
  extends perfectGasI (nIsState=false,
                       pIsState=true,
                       TIsState=true);
equation
  // Interactive change of the state variables
  when CKstate>0.5 and     pre(CKstateIs0) or
       CKstate<0.5 and not pre(CKstateIs0) then
    CKstateIs0 = CKstate < 0.5;
    reinit(p,  Istate.signal[2]);
    reinit(T,  Istate.signal[3]);
  end when;
end perfectGasSS1;

model perfectGasSS2
  extends perfectGasI (nIsState=true,
                       pIsState=false,
                       TIsState=true);
equation
  // Interactive change of the state variables
  when CKstate>0.5 and     pre(CKstateIs0) or
       CKstate<0.5 and not pre(CKstateIs0) then
    CKstateIs0 = CKstate < 0.5;
    reinit(n,  Istate.signal[1]);
    reinit(T,  Istate.signal[3]);
  end when;
end perfectGasSS2;

model perfectGasSS3
  extends perfectGasI (nIsState=true,
                       pIsState=true,
                       TIsState=false);
equation
  // Interactive change of the state variables
  when CKstate>0.5 and     pre(CKstateIs0) or
       CKstate<0.5 and not pre(CKstateIs0) then
    CKstateIs0 = CKstate < 0.5;
    reinit(n,  Istate.signal[1]);
    reinit(p,  Istate.signal[2]);
  end when;
end perfectGasSS3;

model perfectGasInteractive
  Modelica.Blocks.Interfaces.InPort Iparam  (n=2);
  Modelica.Blocks.Interfaces.InPort Ivar    (n=3);
  Modelica.Blocks.Interfaces.InPort Istate  (n=3);
  Modelica.Blocks.Interfaces.InPort CKparam (n=3);
  Modelica.Blocks.Interfaces.InPort CKvar   (n=3);
  Modelica.Blocks.Interfaces.InPort CKstate (n=3);
  Modelica.Blocks.Interfaces.InPort Enabled (n=3);
  Modelica.Blocks.Interfaces.OutPort O      (n=8);
  Modelica.Blocks.Interfaces.OutPort Release(n=1);
  perfectGasSS1 SS1 (CKparam = CKparam.signal[1],
                     CKvar   =   CKvar.signal[1],
                     CKstate = CKstate.signal[1]);
  perfectGasSS2 SS2 (CKparam = CKparam.signal[2],
                     CKvar   =   CKvar.signal[2],
                     CKstate = CKstate.signal[2]);
  perfectGasSS3 SS3 (CKparam = CKparam.signal[3],
                     CKvar   =   CKvar.signal[3],
                     CKstate = CKstate.signal[3]);
equation
  connect(Iparam, SS1.Iparam);
  connect(Istate, SS1.Istate);
  connect(Ivar,   SS1.Ivar);
  connect(Iparam, SS2.Iparam);
  connect(Istate, SS2.Istate);
  connect(Ivar,   SS2.Ivar);
  connect(Iparam, SS3.Iparam);
  connect(Istate, SS3.Istate);
  connect(Ivar,   SS3.Ivar);
  Release.signal = {4.0};
  O.signal = if Enabled.signal[1] > 0.5
             then SS1.O.signal
             else if Enabled.signal[2] > 0.5
                  then SS2.O.signal
                  else if Enabled.signal[3] > 0.5
                       then SS3.O.signal
                       else zeros(size(O.signal, 1));
end perfectGasInteractive;
```