Tesis Doctoral

Doctoral Dissertation

Modelado Orientado a Objetos de Laboratorios Virtuales para la Educación en Control Automático

Object-Oriented Modeling of Virtual Laboratories for Control Education

Carla Martín Villalba

Ingeniera en Electrónica

Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Departamento de Informática y Automática

Madrid, 2007

Departamento	Informática y Automática
Título de la Tesis	Modelado Orientado a Objetos de Laboratorios Virtuales para la Educación en Control Automático
Autor	Carla Martín Villalba
Titulación	Ingeniera en Electrónica Universidad Complutense de Madrid
Directores	Sebastián Dormido Bencomo Alfonso Urquía Moraleda

Department	Informática y Automática
Dissertation Title	Object-Oriented Modeling of Virtual Laboratories for Control Education
Author	Carla Martín Villalba
Academic Degree	Electrical Engineering Universidad Complutense de Madrid
Advisors	Sebastián Dormido Bencomo Alfonso Urquía Moraleda

Acknowledgements

I would like express my deep gratitude to all those who contributed in some way to this thesis.

My supervisor Alfonso Urquía Moraleda for guiding me, encouraging me to do always my best and not less and helping me whenever I needed it.

My supervisor Sebastián Dormido Bencomo for the advices, support and material and financial resources he has provided throughout the course of this work.

Genoveva Martínez and Miguel Sancho for believing in me and giving me my first fellowship to start my research work.

My colleagues of the Computer Science and Automatic Control department for the good atmosphere to develop my work. Especially to Rocío Muñoz and José Manuel Díaz for his friendship and support. I would like to thank the colleagues with whom I shared the "fellowship room" for creating a fun and stimulating environment: Carlos Hernández, Luis Torres, Roselvi Pérez, Arnoldo W. Fernández and Gonzalo Farias.

Pilar Riego, the department secretary, for assisting me in many different ways.

Yves Piguet, the Sysquake developer, for his assistance with the use of Sysquake.

Francisco Esquembre, the Easy Java Simulation (Ejs) developer, for providing me the last releases of Ejs and for his assistance with the use of Ejs.

Dennis Gillet for accepting me in his group to do part of my research work and for his support during my stay at the Ecole Polytechnique Federale de Lausanne (EPFL).

I would like to thank my parents. They always did their best to raise me, support me, teach me and love me. I also would like to thank my brother and sisters (Rita, Ana, Ernesto and Marta) for always being there.

Especially, I would like to give my thanks to José Luis who helped me patiently during these years and provided me a loving environment.

Contents

List of Figures

Resumen en Castellano

(Abstract in Spanish)

Introducción

Los *laboratorios virtuales* son herramientas útiles para la enseñanza del control automático de procesos. Pueden emplearse para explicar los conceptos básicos del control, para mostrar los problemas desde nuevas perspectivas, y para ilustrar cuestiones relativas al análisis y diseño (Johansson et al. 1998, Wittenmark et al. 1998, Dormido 2004).

El laboratorio virtual se compone de un *modelo* y de una *vista*. El *modelo* es la representación matemática de aquellos aspectos del comportamiento del sistema que son relevantes para el propósito del estudio. La *vista* es la interfaz gráfica entre el usuario del laboratorio virtual y el modelo matemático. Su propósito es doble: proporcionar una representación visual del comportamiento del modelo simulado y facilitar la interacción del usuario con el modelo (*interactividad*).

Atendiendo a la forma en que el usuario puede actuar sobre el modelo, cabe distinguir entre los dos tipos siguientes de interactividad:

- *Interactividad continua*. El usuario puede modificar el valor de las entradas, los parámetros y las variables de estado del modelo en cualquier instante durante la ejecución de la simulación. De este modo, el usuario percibe instantáneamente cómo esos cambios afectan a la dinámica del modelo.

– *Interactividad discontinua.* El usuario puede fijar el valor de los parámetros y el estado inicial del modelo, iniciándose entonces automáticamente una réplica de la simulación. Durante la ejecución de la simulación, no se permite al usuario interactuar con el modelo. Una vez finalizada la simulación, sus resultados son mostrados y analizados, permitiéndose entonces al usuario interaccionar de nuevo con el modelo a fin de fijar un nuevo conjunto de valores para los parámetros y las condiciones iniciales.

Existen varias herramientas software cuyo propósito es facilitar la creación de laboratorios virtuales. Dos de ellas son *Sysquake* e *Easy Java Simulations* (Ejs). *Sysquake* (Sysquake 2004, Piguet et al. 1999) es un entorno similar a Matlab, especialmente concebido para el desarrollo de laboratorios virtuales con interactividad discontinua. Ejs (*EJS* 2007, Esquembre 2004) es una herramienta, de código abierto, para el desarrollo de laboratorios virtuales con interactividad continua.

Estas herramientas software permiten crear de un modo sencillo la interfaz gráfica interactiva de usuario (la *vista* del laboratorio virtual). Sin embargo, las capacidades para la definición del modelo y los solucionadores numéricos que proporcionan estas herramientas no se corresponden al estado del arte.

Modelica (Modelica 2005, *Modelica* 2007) es un lenguaje de modelado gratuito, orientado a objetos, que facilita el paradigma del modelado físico (Åström et al. 1998). Los modelos se describen matemáticamente mediante ecuaciones diferenciales, algebraicas y discretas (modelos *DAE híbridos*). El lenguaje Modelica facilita una descripción declarativa (no causal) del modelo, lo cual facilita una mejor reutilización de los modelos. A consecuencia de todo ello, el uso de Modelica reduce considerablemente el esfuerzo de modelado.

El lenguaje Modelica está concebido para su aplicación al modelado de sistemas multi-dominio (por ejemplo, con una parte eléctrica, otra mecánica, otra hidráulica, otra termo-fluida, etc), resultando idóneo para describir el tipo de modelo multi-dominio usado en el control automático.

El desarrollo de librerías de modelos en Modelica es una línea de investigación muy activa, encontrándose disponibles en la actualidad librerías de componentes,

tanto comerciales como gratuitas, para el modelado en los dominios eléctrico, mecánico, termo-fluido, hidráulico, físico-químico, etc., así como librerías que soportan el modelado mediante formalismos tales como las redes de Petri, los grafos de ligadura (bond graphs), etc.

Sin embargo, ni el lenguaje Modelica, ni los entornos de simulación que soportan Modelica (OpenModelica (*OpenModelica* 2007, Fritzson et al. 2002, 2006), Dymola (Dynasim 2006), etc.), ofrecen capacidades para la simulación interactiva. Por tanto, la aplicación de Modelica al desarrollo de laboratorios virtuales es un campo de investigación abierto, en el cual se ha centrado la presente tesis.

Objetivos

Se han planteado básicamente cuatro objetivos en el trabajo de investigación descrito en esta tesis, que son descritos a continuación.

El **primer objetivo** de esta tesis es explorar la viabilidad de usar el lenguaje Modelica en el desarrollo de laboratorios virtuales adecuados para la enseñanza del control automático de procesos. La motivación es conseguir, mediante el empleo del lenguaje Modelica, reducir el esfuerzo requerido para el desarrollo de los laboratorios virtuales.

Se plantea como objetivo soportar el desarrollo de laboratorios virtuales con interactividad continua y con interactividad discontinua, proponiéndose para ello:

1. La implementación de laboratorios virtuales con *interactividad discontinua* combinando el uso de Sysquake y Modelica/Dymola. El modelo del laboratorio virtual se describe usando el lenguaje Modelica y se traduce usando Dymola. La vista del laboratorio virtual se desarrolla usando Sysquake.

   Para poner en práctica esta aproximación, se propone el diseño y programación de una interfaz entre Sysquake y Dymosim. Dymosim es el fichero ejecutable generado por Dymola para el modelo en Modelica. El propósito

de esta interfaz es sincronizar la ejecución de la aplicación de Sysquake y Dymosim.

2. La implementación de laboratorios virtuales con interactividad continua, primero combinando el uso de Ejs y Modelica/Dymola, y finalmente usando sólo Modelica/Dymola. Esto implica:

   (a) Proponer una clasificación de las magnitudes interactivas y analizar las restricciones que la formulación del modelo matemático impone en la selección de las magnitudes interactivas.

   (b) Proponer una metodología sistemática para adaptar cualquier modelo escrito en Modelica a una formulación que permita su simulación con interactividad continua.

   (c) Demostrar la viabilidad de combinar el uso de Ejs y Modelica/Dymola, lo cual se lleva acabo usando las interfaces entre Ejs y Matlab/Simulink y entre Matlab/Simulink y Dymola que han sido desarrolladas por otros autores.

   (d) Diseñar y programar una librería en lenguaje Modelica que facilite la descripción de la vista del laboratorio virtual y que, a partir de dicha descripción, genere automáticamente el código ejecutable de la vista y la comunicación entre el modelo y la vista. El uso de esta librería, que se denominará *VirtualLabBuilder*, permitirá describir el laboratorio virtual usando sólo el lenguaje Modelica.

El **segundo objetivo** de este trabajo de tesis es traducir al lenguaje Modelica, y adaptar para la simulación interactiva, la librería JARA (Urquia 2000). Esta librería contiene modelos de algunos de los principios físico-químicos que encuentran aplicación en el modelado de procesos de transporte, termo-fluidos, cambios de fase y químicos, etc., en el contexto del control automático. La motivación detrás de este objetivo es obtener una librería en lenguaje Modelica que pueda ser usada en el desarrollo de laboratorios virtuales para la enseñanza del control automático de procesos químicos.

El **tercer objetivo** de este trabajo de tesis es desarrollar un conjunto de laboratorios virtuales para la enseñanza del control automático. Algunos de estos laboratorios virtuales serán construidos usando modelos de plantas incluidas en la librería JARA: un reactor químico, un evaporador industrial y un intercambiador de calor.

Finalmente, el **cuarto objetivo** es demostrar que la metodología propuesta y el software programado para el desarrollo de laboratorios virtuales usando sólo Modelica/Dymola se puede aplicar con éxito a:

1. *La solución de un problema industrial real.* Se desarrollará un laboratorio virtual con aplicación al diseño y optimización de una lavadora industrial, colaborando para ello con ingenieros del Departamento de Ingeniería Mecánica del Centro de Investigación Tecnológica IKERLAN (Mondragón, España).

2. *La implementación de un laboratorio virtual basado en un modelo en Modelica complejo, de grandes dimensiones, que ha sido desarrollado por otros autores.* Para este propósito, se usará el modelo del comportamiento termodinámico de una casa solar incluido en la librería *BondLib* (Weiner & Cellier 1993, Cellier & Nebot 2005).

Estructura de la tesis

La tesis se ha estructurado en nueve capítulos y tres apéndices, cuyo contenido se describe brevemente a continuación.

Capítulo 1. Comienza con una breve introducción, en la cual se discute la motivación del trabajo de investigación realizado. Se describen los objetivos y la estructura de la tesis. Finalmente, se enumeran las publicaciones a las que ha dado lugar este trabajo de investigación, así como la participación en proyectos de investigación subvencionados.

Capítulo 2. Se presenta una breve revisión de los conceptos que juegan un papel fundamental en la línea de investigación desarrollada en esta tesis. En

particular, se discute el estado del arte en modelado y simulación, y el desarrollo de laboratorios virtuales, todo ello en el contexto del control automático. Además, se describen algunas características relevantes de la librería JARA.

Capítulo 3. Se propone una metodología para la implementación de laboratorios virtuales combinando el uso de Sysquake y Modelica/Dymola, y se aplica al desarrollo de varios laboratorios virtuales para la docencia del control automático. En el **Apéndice A** se proporciona información adicional sobre el software desarrollado: la librería de funciones LME denominada *sysquakeDymosimInterface*.

Capítulo 4. Se identifican diferentes tipos de magnitudes interactivas y se analizan las ligaduras que el modelo matemático impone sobre la selección de las magnitudes interactivas. Sobre la base de esta discusión, se propone una metodología de modelado para adaptar cualquier modelo escrito en Modelica a una formulación válida para su simulación interactiva *continua*. En el **Apéndice B** se muestra el listado de dos modelos empleados para ilustrar la aplicación de la metodología.

Capítulo 5. Se propone un procedimiento para desarrollar laboratorios virtuales combinando el uso de Ejs, Matlab/Simulink y Modelica/Dymola, y se aplica al desarrollo de varios laboratorios virtuales para la docencia del control automático.

Capítulo 6. Se propone un procedimiento para desarrollar laboratorios virtuales usando sólo Modelica/Dymola, y se discute la estructura y uso de la herramienta software que ha sido diseñada y programada para aplicarlo: la librería en lenguaje Modelica denominada *VirtualLabBuilder*. Finalmente, se describe el desarrollo de varios laboratorios virtuales aplicando los procedimientos propuestos y las herramientas software programadas.

El **Apéndice C** contiene el manual de referencia de la librería *VirtualLabBuilder*, tal como ha sido generado por la herramienta Dymola a partir de la estructura y documentación de la librería.

Capítulo 7. Se discuten los detalles más relevantes del desarrollo de la librería *VirtualLabBuilder*, proporcionando las claves para la extensión de la misma con nuevas clases.

Capítulo 8. Se discute el desarrollo, empleando la librería *VirtualLabBuilder*, de un laboratorio virtual del comportamiento termodinámico de una casa solar.

Capítulo 9. Se presentan las conclusiones del trabajo de investigación y algunas posibles líneas futuras de investigación.

Publicaciones y proyectos de investigación

El trabajo de investigación descrito en la presente Tesis Doctoral ha dado lugar a las publicaciones citadas a continuación.

1. Carla Martín; Alfonso Urquía; Sebastián Dormido (2007): "Implementation of Interactive Virtual Laboratories for Control Education Using Modelica", In: proceedings of *European Control Conference 2007*, Kos (Greece), paper #WeA05.1, pp. 2679-2686.

2. Carla Martín-Villalba; Alfonso Urquía; Sebastián Dormido (2007): "Desarrollo de Laboratorios Virtuales con Aplicación a la Enseñanza del Control usando Modelica", In: proceedings of *V Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática (EIWISA'07), Segundo Congreso Español de Informática (CEDI)*, Zaragoza (Spain).

3. Carla Martín; Alfonso Urquía; Sebastián Dormido (2007): "Virtual-lab of a Solar House Implemented using VirtualLabBuilder Modelica Library", In: proceedings of *Conference on Systems and Control (CSC'2007)*, Marrakech (Morocco), paper #130.

4. Carla Martín; Alfonso Urquía; Sebastián Dormido (2006): "An Approach to Virtual-Lab Implementation using Modelica", In: proceedings of *European Simulation and Modelling Conference (ESM'2006)*, Toulouse (France), pp. 137-141.

5. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Object-Oriented Modeling of Virtual Laboratories for Control Education", In: proceedings of 16^{th} *IFAC World Congress*, Prague (Czech Republic), Paper code: Th-A22-TO/2.

6. Carla Martín; Rocío Muñoz; Alfonso Urquía; Sebastián Dormido (2005): "A Distance Learning Course on Virtual-lab Implementation for High School Science Teachers", In: proceedings of 6^{th} *International Conference on Virtual University*, Bratislava (Slovak Republic), pp. 3-8.

7. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Modelado Orientado a Objetos de Laboratorios Virtuales con Aplicación a la Enseñanza de Control de Procesos Químicos", In: proceedings of *IV Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática (EIWISA'05), Primer Congreso Español de Informática (CEDI)*, Granada (Spain), pp. 21-26.

8. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Modeling of Interactive Virtual Laboratories with Modelica", In: proceedings of 4^{th} *International Modelica Conference*, Hamburg (Germany), pp. 159-168.

9. Carla Martín; Alfonso Urquía; Sebastián Dormido (2004): "JARA2i - A Modelica Library for Interactive Simulation of Physical-Chemical Processes", In: proceedings of *European Simulation and Modelling Conference*, Paris (France), pp. 128-132.

10. Carla Martín; Alfonso Urquía; José Sánchez; Sebastián Dormido; Francisco Esquembre; Jose L. Guzman; Manuel Berenguel (2004): "Interactive Simulation of Object-Oriented Hybrid Models, by Combined Use of Ejs,

Matlab/Simulink and Modelica/Dymola", In: proceedings of 18^{th} *European Simulation Multiconference*, Magdeburg (Germany), pp. 210-215.

11. Alfonso Urquía; Carla Martín; Sebastián Dormido (2005): "Design of SPICE-Lib: a Modelica Library for Modeling and Analysis of Electric Circuits", *Mathematical and Computer Modelling of Dynamical Systems*, Vol. 11, No. 1, pp. 43-60.

12. Carla Martín; Alfonso Urquía; Sebastián Dormido (2003): "SPICELib - Modeling and Analysis of Electric Circuits with Modelica", In: proceedings of 3^{rd} *International Modelica Conference*, Linkoping (Sweden), pp. 161-170.

Trabajos en proceso de revisión

Los siguientes trabajos se encuentran en proceso de revisión:

1. Carla Martín; Alfonso Urquía; Sebastián Dormido: "Object-Oriented Modelling of Virtual-Labs for Education in Chemical Process Control", submitted for publication in *Computer Chemical Engineering*, Elsevier.

2. Carla Martín-Villalba; Alfonso Urquía; Sebastián Dormido: "An Approach to Virtual-Lab Implementation using Modelica", submitted for publication in *Mathematical and Computer Modelling of Dynamical Systems*, Taylor & Francis.

3. Carla Martín; Alfonso Urquía; Sebastián Dormido: "Educación a Distancia del Profesorado de Ciencias en el Desarrollo de Laboratorios Virtuales", submitted for publication in *Revista Iberoamericana de Educación a Distancia (RIED)*, AIESAD.

4. Carla Martín-Villalba; Félix Martínez; Alfonso Urquía; Sebastián Dormido: "Implementation in Modelica of a Virtual-Lab for Testing Washing Machine Designs", regular paper submitted for the *European Simulation and Modelling Conference 2007*.

Proyectos de investigación

La mayoría de los resultados desarrollados en la tesis doctoral han sido obtenidos en el marco de diferentes proyectos de investigación:

1. "Control de sistemas complejos en la logística y producción de bienes y servicios. Acrónimo: COSICOLOGI-CM", *IV PRICIT 2005-2008. Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid. Ref. S-0505/DPI/0391*, Enero 2005 - Diciembre 2008, Investigador principal: Prof. Dr. Sebastián Dormido Bencomo.

2. "Herramientas interactivas para el modelado, visualización, simulación y control de sistemas dinámicos", *CICYT, DPI 2004-01804*, Enero 2004 - Diciembre 2006, Investigador principal: Prof. Dr. Sebastián Dormido Bencomo.

3. "Laboratorios virtuales y remotos de control automático: análisis, diseño y desarrollo", *CICYT, DPI 2001-01012*, Enero 2002 - Diciembre 2004, Investigador principal: Prof. Dr. Sebastián Dormido Bencomo.

Conclusiones

Se han propuesto tres enfoques diferentes para el desarrollo de laboratorios virtuales usando el lenguaje Modelica:

1. El desarrollo de laboratorios virtuales con interactividad discontinua, combinando el uso de Sysquake y Modelica/Dymola. Este trabajo se encuentra resumido en (Martin et al. 2005b,c).

2. El desarrollo de laboratorios virtuales con interactividad continua, combinando el uso de Ejs y Modelica/Dymola. El planteamiento y los resultados obtenidos se resumen en (Martin et al. 2004a,b, 2005a,b,c).

3. El desarrollo de laboratorios virtuales con interactividad continua, usando sólo Modelica/Dymola. Este trabajo se encuentra resumido en (Martin et al. 2006, Martin-Villalba et al. 2007, Martin et al. 2007).

Se han propuesto las metodologías y desarrollado las herramientas software necesarias para llevar a la práctica las tres métodos anteriormente expuestos para el desarrollo de laboratorios virtuales. En concreto:

1. Se ha programado una interfaz entre Sysquake y Dymosim. Esta interfaz consiste en un conjunto de funciones en el lenguaje LME, que pueden ser invocadas desde las aplicaciones de Sysquake. Dicha interfaz está disponible en `http://www.euclides.dia.uned.es`

2. Se ha propuesto una metodología para adaptar cualquier modelo escrito en Modelica a una formulación válida para la simulación con interactividad continua. Se han considerado los dos casos siguientes:

   (a) Pueden definirse todas las magnitudes interactivas simultáneamente como variables de estado.

   (b) Lo anterior no es posible, con lo cual es necesario soportar simultáneamente varias selecciones de las variables de estado.

3. Se ha propuesto un procedimiento para desarrollar laboratorios virtuales combinando el uso de Ejs y Modelica/Dymola, valiéndose para ello de las interfaces existentes entre Ejs y Simulink, y entre Dymola y Simulink.

4. Se ha diseñado y programado la librería en lenguaje Modelica denominada *VirtualLabBuilder*, gracias a la cual puede describirse el laboratorio virtual empleando únicamente lenguaje Modelica. Su documentación on-line está disponible en `http://www.euclides.dia.uned.es`

La metodología propuesta para adaptar modelos escritos en Modelica a la simulación interactiva ha sido aplicada con éxito a las dos librerías indicadas a continuación. Ambas pueden ser descargadas de `http://www.euclides.dia.uned.es`

1. La librería JARA ha sido traducida al lenguaje Modelica y ha sido adaptada para la simulación con interactividad continua y discontinua. Esta nueva versión de la librería JARA, en lenguaje Modelica y adecuada para la simulación interactiva, se ha denominado *JARA 2i*.

2. Se ha diseñado y programado en Modelica la librería *tankProcessLAB*, y se ha adaptado para la simulación con interactividad continua y discontinua.

Las metodologías propuestas y las herramientas software programadas han sido aplicadas con éxito al desarrollo de los siguientes laboratorios virtuales para la educación en control de procesos:

– *Laboratorios virtuales con interactividad discontinua.* Laboratorios virtuales de un controlador de histéresis, un reactor químico, un intercambiador de calor de doble tubo y un evaporador industrial.

– *Laboratorios virtuales con interactividad continua.* Laboratorios virtuales de un sistema de cuatro tanques, un evaporador industrial, un reactor químico y un intercambiador de calor de doble tubo.

Finalmente, el trabajo de investigación realizado para posibilitar la descripción de los laboratorios virtuales usando únicamente el lenguaje Modelica ha sido aplicado con éxito:

1. Al desarrollo de un laboratorio virtual para el diseño y análisis de lavadoras con tambor, con aplicación a un problema de diseño industrial real. La definición de las especificaciones del laboratorio virtual y la programación del modelo (en lenguaje Modelica) ha sido realizada por los usuarios del laboratorio virtual: ingenieros del Departamento de Ingeniería Mecánica del Centro de Investigaciones Tecnológicas IKERLAN (Mondragón, Spain).

2. Al desarrollo de un laboratorio virtual que ilustra el comportamiento termodinámico de una casa solar. Con ello se demuestra que los resultados obtenidos son aplicables a modelos de grandes dimensiones y complejidad, desarrollados además por otros autores.

Líneas futuras de investigación

Finalmente, se exponen a continuación algunas ideas sobre posibles extensiones del trabajo de investigación realizado en esta tesis:

- Implementar una herramienta software que permita realizar automáticamente las adaptaciones al modelo para la simulación interactiva que han sido propuestas en esta tesis.
- Desarrollar más elementos gráficos interactivos e incluirlos en la librería *VirtualLabBuilder*. Por ejemplo, elementos de dibujo que describan formas 3-D.
- Adaptar las librerías incluidas en la librería estándar de Modelica a la simulación interactiva y desarrollar los correspondientes elementos gráficos interactivos.
- Explorar el uso de *VirtualLabBuilder* en otros entornos de simulación que soportan Modelica, tales como OpenModelica y DrModelica (Lengquist et al. 2003).
- Extender las capacidades de *VirtualLabBuilder* de modo que la interfaz gráfica interactiva de los laboratorios virtuales sea un applet de Java.

1

Introduction, Objectives and Structure

1.1 Introduction

Virtual-labs are useful tools for control education. They can be used to explain basic concepts, to provide new perspectives of a problem, and to illustrate analysis and design topics (Johansson et al. 1998, Wittenmark et al. 1998, Dormido 2004).

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 graphical 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.

Two alternative types of interactivity can be considered:

- *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.

There exist several software tools specifically intended for the implementation of virtual-labs. Two of them are Sysquake and Easy Java Simulations (Ejs). Sysquake (Sysquake 2004, Piguet et al. 1999) is a Matlab-like environment for developing virtual-labs with batch interactivity. Ejs (*EJS* 2007, Esquembre 2004) is a software tool for developing virtual-labs with runtime interactivity. These software tools allow easy creation of the interactive graphical user interface. However, the model definition capabilities and the numerical solvers provided by these tools are not the state-of-the-art.

Modelica (Modelica 2005, *Modelica* 2007) is a freely available, object-oriented modeling language that facilitates the physical modeling paradigm (Åström et al. 1998). Models are mathematically described by differential, algebraic and discrete equations. The Modelica language 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.

Modelica is intended for multi-domain modeling. Currently, a number of free and commercial component libraries in different domains are available (*Modelica* 2007), 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 (e.g., OpenModelica (*OpenModelica* 2007, Fritzson et al. 2002, 2006), Dymola (Dynasim 2006), etc.) support interactive simulation. As a consequence, extending the Modelica capabilities in order to facilitate the implementation of virtual-labs is an open research field.

1.2 Objectives

The **first objective** of this dissertation is to explore the feasibility of using Modelica language in the implementation of virtual-labs for control education. The motivation behind this objective is to reduce the virtual-lab development effort. Different approaches to this objective are considered:

1. The implementation of virtual-labs with batch interactivity by combining the use of Sysquake and Modelica/Dymola. The virtual-lab model is programmed using Modelica language and translated using Dymola. The graphical user-to-model interface is implemented using Sysquake.

   In order to implement this approach, the design and programming of a Sysquake-to-Dymosim interface is proposed. Dymosim is the executable file generated by Dymola for the Modelica model. The purpose of this interface is to synchronize the execution of Dymosim and the Sysquake application.

2. The implementation of virtual-labs with runtime interactivity, firstly combining the use of Ejs and Modelica/Dymola, and finally using only Modelica/Dymola. This implies:

   (a) To identify different types of interactive quantities and to analyze the constraints that the mathematical model imposes on the selection of the interactive quantities.

   (b) To propose a systematic methodology for adapting any Modelica model to runtime interactive simulation.

   (c) To demonstrate the feasibility of combining the use of Ejs and Modelica/Dymola, which is accomplished by using the Ejs-to-Matlab/Simulink and Matlab/Simulink-to-Dymola interfaces.

   (d) To design and program a Modelica library to facilitate the implementation of the virtual-lab view and the model-to-view communication. The use of this library will allow to describe the virtual-lab using only the Modelica language.

The **second objective** of this dissertation work is translating into Modelica language and adapting for interactive simulation the JARA library (Urquia 2000). This library contains models of some fundamental physical-chemical principles. Its main application domain is the modeling of transport, thermo-fluid, phase-change and chemical processes in the context of automatic control. The motivation behind this objective is to obtain a Modelica library that could be used in the development of virtual-labs for chemical process control education.

The **third objective** of this dissertation work is to develop a set of virtual-labs for process control education. Some of these virtual-labs will be built using models of process plants included in the JARA library: a chemical reactor, an industrial boiler and a double-pipe heat exchanger. Other virtual-labs will be composed by using a Modelica library for modeling basic hydraulic processes that will be developed as a part of this dissertation work.

Finally, the **fourth objective** is to demonstrate that the proposed approach to the implementation of virtual-labs using only Modelica/Dymola can be applied to:

1. The solution of a real industrial problem. A virtual-lab for aiding in the design and optimization of a drum-type washing machine will be developed in cooperation with engineers of the Mechanical Engineering Department of the IKERLAN Technological Research Center (Mondragón, Spain).

2. The implementation of a virtual-lab based on a complex Modelica model that has been developed by other authors. The model of a solar house that is included in the *BondLib* Modelica library (Weiner & Cellier 1993, Cellier & Nebot 2005) will be used for this purpose.

1.3 Structure of the dissertation

This dissertation has been structured as follows:

Chapter 2. A brief review of some concepts that play a fundamental role is this dissertation is presented. In particular, it is discussed the state-of-the-art in modeling, simulation, and virtual-lab implementation in the context of automatic control. Additionally, some relevant characteristics of the JARA Modelica library are described.

Chapter 3. A methodology for the implementation of virtual-labs by combining the use of Sysquake and Modelica/Dymola is proposed. The development of some virtual-labs illustrating this approach is discussed.

Additional information about the developed software (a library of LME functions called *sysquakeDymosimInterface*) is provided in **Appendix A**.

Chapter 4. Different types of interactive quantities are identified and the constraints that the mathematical model imposes on the selection of the interactive quantities are analyzed. On the basis of this discussion, a modeling methodology to adapt any Modelica model for *runtime* interactive simulation is proposed.

The code of two models illustrating the application of the proposed methodology is provided in **Appendix B**.

Chapter 5. A methodology for implementing virtual-labs by combining the use of Ejs, Matlab/Simulink and Modelica/Dymola is proposed. The development of several virtual-labs according to this methodology is described.

Chapter 6. A novel approach to the virtual-lab implementation using only Modelica/Dymola is proposed. The *VirtualLabBuilder* Modelica library is presented and some relevant information about its use is provided. The development of virtual-labs using Modelica language is illustrated by means of some case studies.

The **Appendix C** contains the *VirtualLabBuilder* user's reference. It has been generated by the Dymola tool from the library structure and documentation.

Chapter 7. The most relevant implementation details of the *VirtualLabBuilder* Modelica library are discussed. The extension of the library with additional classes is addressed in this chapter.

Chapter 8. The implementation of a virtual-lab illustrating the thermodynamic behavior of a solar house is discussed. This virtual-lab is described using only the Modelica language.

Chapter 9. The conclusions and the future research are presented.

1.4 Publications

1. Carla Martín; Alfonso Urquía; Sebastián Dormido (2007): "Implementation of Interactive Virtual Laboratories for Control Education Using Modelica", In: proceedings of *European Control Conference 2007*, Kos (Greece), paper #WeA05.1, pp. 2679-2686.

2. Carla Martín-Villalba; Alfonso Urquía; Sebastián Dormido (2007): "Desarrollo de Laboratorios Virtuales con Aplicación a la Enseñanza del Control usando Modelica", In: proceedings of *V Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática (EIWISA'07), Segundo Congreso Español de Informática (CEDI)*, Zaragoza (Spain).

3. Carla Martín; Alfonso Urquía; Sebastián Dormido (2007): "Virtual-lab of a Solar House Implemented using VirtualLabBuilder Modelica Library", In: proceedings of *Conference on Systems and Control (CSC'2007)*, Marrakech (Morocco), paper #130.

4. Carla Martín; Alfonso Urquía; Sebastián Dormido (2006): "An Approach to Virtual-Lab Implementation using Modelica", In: proceedings of *European*

Simulation and Modelling Conference (ESM'2006), Toulouse (France), pp. 137-141.

5. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Object-Oriented Modeling of Virtual Laboratories for Control Education", In: proceedings of 16^{th} *IFAC World Congress*, Prague (Czech Republic), Paper code: Th-A22-TO/2.

6. Carla Martín; Rocío Muñoz; Alfonso Urquía; Sebastián Dormido (2005): "A Distance Learning Course on Virtual-lab Implementation for High School Science Teachers", In: proceedings of 6^{th} *International Conference on Virtual University*, Bratislava (Slovak Republic), pp. 3-8.

7. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Modelado Orientado a Objetos de Laboratorios Virtuales con Aplicación a la Enseñanza de Control de Procesos Químicos", In: proceedings of *IV Jornadas de Enseñanza vía Internet/Web de la Ingeniería de Sistemas y Automática (EIWISA'05), Primer Congreso Español de Informática (CEDI)*, Granada (Spain), pp. 21-26.

8. Carla Martín; Alfonso Urquía; Sebastián Dormido (2005): "Modeling of Interactive Virtual Laboratories with Modelica", In: proceedings of 4^{th} *International Modelica Conference*, Hamburg (Germany), pp. 159-168.

9. Carla Martín; Alfonso Urquía; Sebastián Dormido (2004): "JARA2i - A Modelica Library for Interactive Simulation of Physical-Chemical Processes", In: proceedings of *European Simulation and Modelling Conference*, Paris (France), pp. 128-132.

10. Carla Martín; Alfonso Urquía; José Sánchez; Sebastián Dormido; Francisco Esquembre; Jose L. Guzman; Manuel Berenguel (2004): "Interactive Simulation of Object-Oriented Hybrid Models, by Combined Use of Ejs, Matlab/Simulink and Modelica/Dymola", In: proceedings of 18^{th} *European Simulation Multiconference*, Magdeburg (Germany), pp. 210-215.

11. Alfonso Urquía; Carla Martín; Sebastián Dormido (2005): "Design of SPICE-Lib: a Modelica Library for Modeling and Analysis of Electric Circuits", *Mathematical and Computer Modelling of Dynamical Systems*, Vol. 11, No. 1, pp. 43-60.

12. Carla Martín; Alfonso Urquía; Sebastián Dormido (2003): "SPICELib - Modeling and Analysis of Electric Circuits with Modelica", In: proceedings of 3^{rd} *International Modelica Conference*, Linkoping (Sweden), pp. 161-170.

The revision process of the following manuscripts is on going:

1. Carla Martín; Alfonso Urquía; Sebastián Dormido: "Object-Oriented Modelling of Virtual-Labs for Education in Chemical Process Control", submitted for publication in *Computer Chemical Engineering*, Elsevier.

2. Carla Martín-Villalba; Alfonso Urquía; Sebastián Dormido: "An Approach to Virtual-Lab Implementation using Modelica", submitted for publication in *Mathematical and Computer Modelling of Dynamical Systems*, Taylor & Francis.

3. Carla Martín; Alfonso Urquía; Sebastián Dormido: "Educación a Distancia del Profesorado de Ciencias en el Desarrollo de Laboratorios Virtuales", submitted for publication in *Revista Iberoamericana de Educación a Distancia (RIED)*, AIESAD.

4. Carla Martín-Villalba; Félix Martínez; Alfonso Urquía; Sebastián Dormido: "Implementation in Modelica of a Virtual-Lab for Testing Washing Machine Designs", regular paper submitted for the *European Simulation and Modelling Conference 2007*.

1.5 Research projects

Most of the results developed in the doctoral dissertation have been obtained in the framework of different research projects:

1. "Control de sistemas complejos en la logística y producción de bienes y servicios. Acrónimo: COSICOLOGI-CM", *IV PRICIT 2005-2008. Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid. Ref. S-0505/DPI/0391*, January 2005 - December 2008, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

2. "Herramientas interactivas para el modelado, visualización, simulación y control de sistemas dinámicos", *CICYT, DPI 2004-01804*, January 2004 - December 2006, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

3. "Laboratorios virtuales y remotos de control automático: análisis, diseño y desarrollo", *CICYT, DPI 2001-01012*, January 2002 - December 2004, Principal researcher: Prof. Dr. Sebastián Dormido Bencomo.

2

Object-Oriented Modeling and Interactive Simulation

2.1 Introduction

A brief review of the state-of-the-art in modeling, simulation, and virtual-lab implementation in the context of automatic control is presented. Firstly, the historical development of continuous-time modeling paradigms and simulation tools is discussed. Secondly, some relevant features of the object-oriented modeling languages and environments are described. Special attention is paid to the Modelica language and the Dymola environment, because they play a fundamental role in this work. Thirdly, some of the many benefits of virtual-labs for control education and some key characteristics of four software tools intended for virtual-lab implementation are discussed. These tools are LabVIEW, Sysquake, Easy Java Simulations and OOCSMP. Finally, the previous work on virtual-lab implementation using Modelica that has been developed by other authors is described.

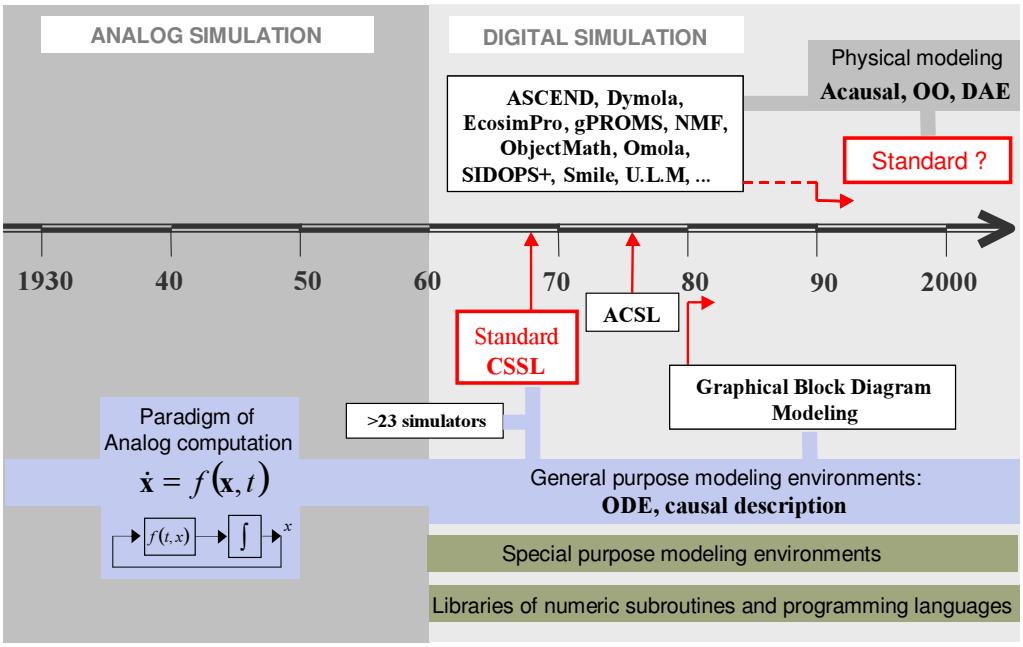


Figure 2.1: Evolution of continuous-time modeling and simulation.

2.2 Evolution of continuous-time modeling and simulation

Graphical block diagram modeling is widely used in control engineering (Karayanakis 1995). Some examples of languages and environments supporting this paradigm are Matlab/Simulink (*Matlab* 2007), MATRIX$_X$/SystemBuild (Shah et al. 1985) and ACSL Graphics Modeller (MGA Software 1996). *Block diagram modeling paradigm* might be considered as a heritage of analog simulation (Åström et al. 1998).

On the other hand, object-oriented modeling languages and compilers supporting the *physical modeling paradigm* have become available since the 1990's decade. This is driven by demands from users to be able to simulate complex multi-domain models.

In order to put into their historical context these modelling paradigms, some aspects of the evolution of continuous-time modeling and simulation are outlined next.

2.2.1 Analog simulation

Analog techniques were predominant from 1920 to 1950 (see Figure 2.1). The idea is to model a system in terms of ordinary differential equations (ODE) and then make a physical device that obeys the equations. The physical system is initialized with proper initial values and its development over time then mimics the differential equation (Åström et al. 1998).

First analog simulators were mechanical systems. The mechanical differential analyzer developed by Vanevar Bush at MIT was the first general purpose tool to simulate dynamical systems (Bush 1931). A major shift in technology occurred in 1947, when it was demonstrated that simulation could be done electronically (Ragazzini et al. 1947).

Variables were represented as voltages in the electronic simulators. The differential equations were represented in terms of the fundamental operations: addition, multiplication, integration and function generation. Since the analog computer has limited range and resolution, the variables must be scaled (Jackson 1960). Several manual steps could be required to transform the model equations into the ODE formulation (i.e., $\frac{dx}{dt} = f(t, x)$). These formula manipulations, which are tedious and error prone, include breaking the algebraic loops, for instance by including small capacitors (Åström et al. 1998).

2.2.2 The CSSL standard

The use of digital computers in simulation was explored since the advent of computes in the early 1950's. This development was triggered by Selfridge, which showed how a digital computer can emulate a differential analyzer (Selfridge 1955). By 1967, there were more than 23 programs for model simulation.

The simulation paradigm adopted by these programs was the same used in analog simulators, i.e., to describe the model in terms of ordinary differential equations (ODE), which were solved using numerical integration techniques. ODE solvers are based on the idea of replacing the differential equations by difference equations. Methods well known in the 1960's include Euler method,

Runge-Kutta methods and explicit multi-step methods. Important contributions were given to stability of difference approximations (Dahlquist 1959, Henrichi 1962). The automatic step length adjustment was another important contribution (Fehlberg 1964). However, ODE solvers well suited for stiff systems were not available at that time.

The CSSL standard appeared at 1967 (Augustin et al. 1967). A system can be described in CSSL language in three different ways: (1) as an interconnection of blocks; (2) by mathematical expressions; and (3) by conventional programming constructs as in FORTRAN.

CSSL defines a set of operators. For instance, INTEG emulates the integrator of the analog computer, and IMPL allows breaking the algebraic loops. The user can define new block types by means of a MACRO definition. Additionally, CSSL contains sentences to select integration routines and their parameters, control the simulation and document the results.

Software products based on the CSSL definition appeared. One example is ACSL (Mitchell & Gauthier 1976), which was for a long time a "de facto" standard for simulation. Constructors for combining continuous/discrete modeling were later added to ACSL. ACSL Graphics Modeller was introduced in 1993, supporting the graphical block diagram modeling.

2.2.3 Graphical block diagram modeling

This modeling paradigm facilitates a hierarchical and modular description of the model. The model is built from graphical blocks, which have input and output ports. The connection among the blocks is performed by connecting these ports.

The analog computing paradigm with its requirement of explicit state models (ODE) is a fundamental limitation of the block diagram modeling (Åström et al. 1998). The blocks have a unidirectional data flow, from input to output. As a consequence, it is cumbersome to build physics-based model libraries in the block diagram languages.

Some tools supporting graphical block diagram modeling are Simulink (originally called Simulab) (Grace 1991), Scicos (Bunks et al. 1999, Chancelier et al.

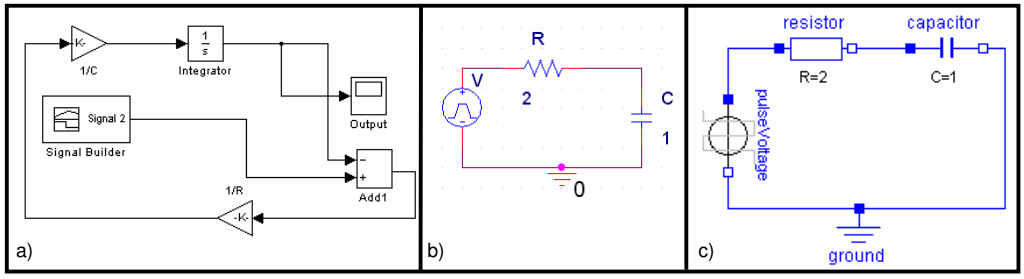


Figure 2.2: RC circuit model implemented using: a) Simulink; b) PSpice; and c) Modelica/Dymola.

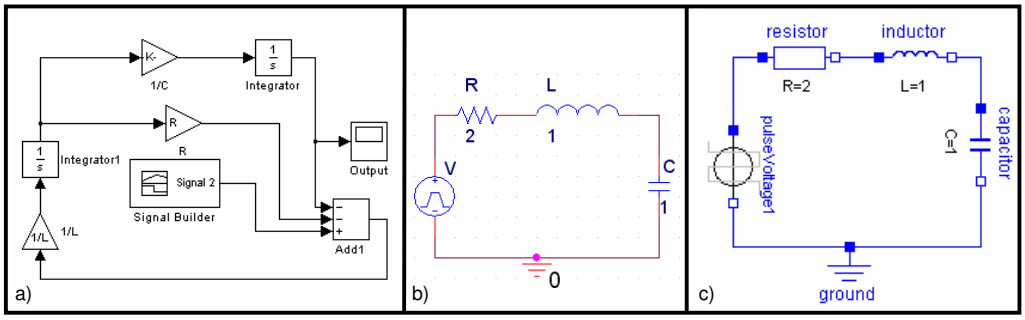


Figure 2.3: RLC circuit model implemented using: a) Simulink; b) PSpice; and c) Modelica/Dymola.

2002) and SystemBuild (Shah et al. 1985). These tools are integrated in the matrix environments Matlab (*Matlab* 2007), Scilab (*Scilab* 2007) and MATRIX$_X$ (*MATRIX$_X$* 2007), respectively.

Two models of electric circuits are used to illustrate how models are described according to different modeling paradigms. The RC circuit shown in Figure 2.2a has been implemented using Matlab/Simulink. In order to build this model, the following steps were taken:

1. The equations for each element of the circuit were derived:

$$V = pulse(t) \tag{2.1}$$

$$v_R = i \cdot R \tag{2.2}$$

$$C \cdot \frac{dv_C}{dt} = i \tag{2.3}$$

$$V = v_R + v_C \tag{2.4}$$

2. The computational causality of the model was calculated, and the model equations were manipulated. The variable on the left side is each equation is

the variable to be calculated from the equation. The manipulated equations are the following:

$$V = pulse(t) \tag{2.5}$$

$$\frac{dv_C}{dt} = \frac{i}{C} \tag{2.6}$$

$$i = \frac{V - v_C}{R} \tag{2.7}$$

3. These equations were transformed into the block description shown in Figure 2.2a.

Connecting in series an inductor, the circuit shown in Figure 2.3 is obtained. The computational causality of the model needs to be re-calculated. The manipulated model is the following:

$$V = pulse(t) \tag{2.8}$$

$$\frac{dv_C}{dt} = \frac{i}{C} \tag{2.9}$$

$$\frac{di}{dt} = \frac{V - i \cdot R - v_C}{L} \tag{2.10}$$

The model described using Matlab/Simulink is shown in Figure 2.3a.

2.2.4 Modeling in specific domains

There are modeling environments that allow the user to compose models in specific domains. A model is assembled simply by connecting components from predefined libraries. Some examples of specific domain simulators are the following:

- PSpice (Nagel & Pederson 1973, Nagel 1975, Kielkowski 1998, OrCAD Inc. 1999) and VHDL-AMS (IEEE 1997) for electronic systems.
- ADAMS (*Adams* 2007) and SIMPACK (*SIMPACK* 2007) for mechanical systems.
- gPROMS (Barton & Pantelides 1994) for energy and process systems.

The models of the RC and RCL circuits composed using PSpice are shown in Figures 2.2b and 2.3b. The model description is very similar to the schematic diagram of the circuit.

2.2.5 Physical modeling

The physical modeling paradigm is based on the modular modeling methodology. Typically, the basic stages of the physical modeling are (Åström et al. 1998):

1. Definition of the system structure and partition of the system into subsystems.

2. Definition of the interaction among the subsystems.

3. Description of the internal behavior of each subsystem, independently of each other, in terms of mass, energy and momentum balances and of material equations.

The modeling knowledge is represented as differential, algebraic and discrete equations that may change by being triggered by events (i.e., hybrid models). A model is considered as a constraint between system variables (Åström et al. 1998).

In order to perform the design of a dynamic system, we have to define the structure of the system, identify its different parts and the interactions between them. Then, the internal behavior of each part is defined indepently. A language that supports object oriented modeling of hybrid dynamic systems require a syntax suitable for the definition, parametrization, reuse, connection and instantiation of classes. The syntax has to facilitate the information encapsulation.

The first languages supporting physical modeling appeared by the mid 1980's (Åström et al. 1998). Among the first languages supporting the physical modeling paradigm were Dymola (Elmqvist 1978) and Omola (Andersson 1989*a*,*b*, 1990, 1994). Other object-oriented modeling languages are ABACUSS II (*ABACUSS II* 2007), ASCEND (Piela 1989), Smile (Kloas et al. 1995), gPROMS (Barton & Pantelides 1994), MODE.LA (*MODE.LA* 2007, Stephanopoulos et al. 1990),

ObjectMath (Fritzson et al. 1995), EcosimPro Language (Empresarios Agrupados 2007*a*,*b*,*c*) and Modelica (Modelica 2005, *Modelica* 2007). Among the first publications concerning interactive simulation is (Korn 1989).

The common characteristics of these modeling languages are the object-oriented, non-causal modeling methodology and the need for automatic symbolic formula manipulation. Object-oriented modeling is based, among others, on three principles: abstraction, encapsulation and modularity. Object-oriented modeling languages support a declarative description of the model, based on equations (i.e., equation-oriented modeling) instead of assignment statements. The information of what variable to solve for in each equation is not included in the model (i.e., non-causal modeling). This permits better reuse of models since equations do not specify a certain data flow direction. Thus a model can adapt to more than one data flow context. The software tools supporting these modeling languages implement algorithms to automatically decide which equation to use for calculating each unknown variable.

The symbolic manipulations that these software tools carry out on the model can be classified into two types according to their purpose.

1. Manipulations intended to translate the object-oriented description of the model into the so-called flat model (Fritzson et al. 2002, Fritzson 2004). The flat model contains the complete set of model equations and functions, with all the object-oriented structure removed.

2. Manipulations intended to transform the flat model into an efficiently solvable from. This second type of manipulations includes (Cellier 1991, Cellier & Kofman 2006, Fritzson 2004):

   – The efficient formulation of the complete-model equations, eliminating the redundant variables and the trivial equations resulting from the submodels connections (Elmqvist 1978, Bunus & Fritzson 2002).

   – The sorting of the equations (Elmqvist 1978, Cellier & Kofman 2006).

   – The symbolic manipulation of those equations in which the unknown variable appears linearly.

– The reduction of the system index to zero or one (Brenan et al. 1996, Mattsson & Soderlind 1992, Pantelides 1988).

The modeling environments need, for simulating hybrid models (i.e., a set of synchronous differential, algebraic and discrete equations), the following:

1. A simulation algorithm appropriate for hybrid systems (for instance, the Omola simulation algorithm is described in (Andersson 1994)).

2. An adequate treatment of the discrete events (Elmqvist et al. 1993): the detection, the accurate determination of the trigger time (Cellier 1979, Cellier et al. 1993, Elmqvist et al. 1993, 1994) and the re-start problem solution.

3. Algorithms to carry out the symbolic manipulation of the linear systems of simultaneous equations and to tear the nonlinear ones (Elmqvist & Otter 1994).

In addition, the modeling environment needs to include at least one DAE-solver algorithm (Gear 1971, Brenan et al. 1996, Hairer et al. 1989), for instance, DASSL (Brenan et al. 1996). The simulation efficiency is notably increased with the use of inline integration algorithms (Elmqvist et al. 1995).

2.3 Modelica language

Modelica is an object-oriented modeling language based on the physical modeling paradigm (Modelica 2005, *Modelica* 2007). Modelica language has been designed by the developers of the object oriented languages ALLAN (Jeandel et al. 1997), Dymola (Dynasim 2006), NMF (Sahlin et al. 1996), ObjectMath (Fritzson et al. 1995), Omola (Andersson 1989*a*,*b*, 1990, 1994), SIDOPS+ (Breuneuse & Broenink 1997), Smile (Kloas et al. 1995) and a number of modeling practitioners in different domains. Modelica is intended to serve as a standard format so that models arising in different domains can be exchanged between tools and users.

Modelica supports multi-domain modeling and several formalisms, such as ODE, DAE, bond graphs (Karnopp & Rosenberg 1968, Karnopp et al. 1990, Thoma 1990, Cellier 1991), finite state automata, and Petri nets. In addition, PDE support in Modelica is an open research field (Saldamli 2002, 2005, 2006).

A number of free and commercial component libraries in different domains are available (*Modelica* 2007), including electrical (Clauss et al. 2000, Cellier & Nebot 2005, Urquia et al. 2005, Martin et al. 2003) mechanical (Otter et al. 2003), thermo-fluid (Eborn 1998, 2001, Tummescheit 2002, Elmqvist et al. 2003, Casella & Leva 2003, 2006, Mattsson 1997), physical-chemical (Urquia & Dormido 2003), bond graph (Cellier & Nebot 2005, Zimmer & Cellier 2006) and state machines (Otter et al. 2005).

Some features of the Modelica language version 2.2 are described below (*Modelica* 2007, Modelica 2005, Fritzson 2004, Fritzson & Engelson 1998). The basic structuring element in Modelica is a class. There are seven restricted class categories with specific keywords: *type*, *connector*, *model*, *package*, *block*, *function* and *record*.

Partial classes. The class prefix *partial* is used to indicate that a class is incomplete and cannot be instantiated. Models are classes of type *model* or *partial model*. Classes of type *model* describe a complete model, whereas those of type *partial model* describe only certain model properties and cannot be instantiated.

Package. Classes can be grouped in special classes, named *package*. *Packages* contain only constant and classes declarations. The classes contained in the *package* can be accessed using the dot notation.

Reuse. Modelica allows class reuse in the two following ways:

- Reusing the classes through *composition*. New values can be set to its parameters. There is a type of class, named *record*, whose purpose is to group a set of parameters.
- Reusing the classes through *inheritance*. When a class composed by other classes is inherited it is possible, unless it is forbidden on purpose,

to modify the class and the value of the parameters of each submodel composing the inherited class (*redeclare* sentence). The new class of the submodel must be a subtype of the older one (class A is a subtype of class B if class A includes all the public components of class B). Modelica support multiple inheritance.

Replaceable classes. Additionally, the class of some submodels (*replaceable model*) and/or connectors (*replaceable connector*) that compose a class can be declared as parameters that can be redefined when the class is reused.

Information encapsulation. Modelica hides the information contained in the section *protected* of a class when it is reused as a submodel. The rest of the variables can be accessed using the dot notation. The variables contained in the *protected* section of a class can be accessed from any other class that extends this class.

Class interface. Interface variables can be *flow* (its sum is 0 at the connection point) or *non-flow* (are equal at the connection point). These variables are gathered in special classes named *connector*. *Connector* classes can't contain equations. Classes that describe models inherit their interface description. The connection between two submodels is defined by applying the *connect* function to a couple of classes of type *connector*. The computation causality of the terminal variables can be set by using the prefixes *input* and *output*. Modelica checks that the computational causality is the one set by using these two prefixes.

Types. The basic predefined built-in types of Modelica are *Real*, *Integer*, *Boolean*, *String* and the basic *enumeration* type. New types can be defined and extended, with the restriction that type classes cannot include variables and equations.

Blocks. A specific type of class named *block* is defined to describe block diagrams. The terminal variables of the block diagrams have a fixed computational causality.

Regular structures. Set of equations, submodels and connections can be defined using *for* loops.

Algorithms. Modelica allows to define in a class a sorted sequence of assignations by including them in a special section (*algorithm*). The *algorithm* section can contain assignations of the type $\langle variable \rangle := \langle expression \rangle$, *for* and *while* structures.

Functions. There are special classes named *function*, that can include local and global variables and an *algorithm* section. Local variables are defined inside a *protected* section. Global variables can only be defined as computational input or output (these are marked in the code by keywords input and output). Class of type *function* can encapsulate calls to functions defined in other languages.

Built- in operators. Built-in operators of Modelica have the same syntax as a function call. However, the result of a built-in operator depends not only on the input arguments but also on the status of the simulation. Some Modelica built-in operators are the following:

- $der(expr)$: performs the time derivative of the expression $expr$. The expression $expr$ need to be a subtype of *Real* and the expression and all its sub-expressions must be differentiable. If $expr$ is an array, the operator is applied to all elements of the array.
- $assert(condition, message)$: allows to show an error message when the value of the boolean expression *condition* is *false*.
- $pre(y)$: returns the "left limit" of variable $y(t)$ at a time instant t.
- $reinit(x, expr)$: reinitializes the value of the state variable x with $expr$ at an event instant. It can only be applied in the body of a when-clause.
- $initial()$: returns *true* during the initialization phase and *false* otherwise.
- $terminal()$: returns true at the end of a successful analysis.
- $terminate(message)$: successfully terminates the current analysis.

– $sample(start, interval)$: returns *true* and triggers time events at time instants $start + i * interval$, where $i = 0, 1, ...$ During continuous integration the operator returns always *false*.

Variable structure and discrete events. Modelica provides the *if–then–else* structure to describe variable structure models.

The instantaneous equations are modeled using the *when* structure. The expression of a *when* clause shall be a discrete-time *Boolean* scalar or vector expression. The equations and algorithm statements within a *when* clause are activated when the scalar or any one of the elements of the vector expression becomes *true*.

Inner and outer prefixes. An element declared with the prefix *outer* references an element instance with the same name and the prefix *inner*. There shall exist at least one corresponding *inner* element declaration for an *outer* element reference.

Initialization. The model initialization takes place just before the simulation starts. The *initial algorithm* and *initial equation* sections are executed during the initialization phase. The *initial algorithm* section can include any kind of equation except *when*-statements. The *initial equation* section can include any kind of equation except *when*-equations. The equations inside a *when* are included in the initialization equation system only if they are explicitly enable with the *initial()* operator. Additionally, it is possible to specify the initial value of a variable through its *start* attribute.

Selection of the state variables. Modelica supports the user's control on the state variables selection, via the *stateSelect* attribute of *Real* variables (Otter & Olsson 2002). This attribute values include "never" (the variable will never be selected as state variable) and "always" (the variable will always be used as a state).

Annotations. Annotations are intended for storing extra information about a model, such as the model icon representation, the structure of composed

models and connection between submodels, documentation or versioning, etc.

2.4 Modelica simulation environments

OpenModelica (*OpenModelica* 2007, Fritzson et al. 2002, 2006), Dymola (Dynasim 2006) and MathModelica System Designer (*MathModelica* 2007) are three modeling and simulation environments that support the Modelica language. OpenModelica environment is free, and it can be used (from version 1.4.2) together with the graphical editor MathModelica Lite. On the other hand, Dymola and MathModelica System Designer are commercial environments.

The simulation environment used in this dissertation is Dymola. Dymola translates the Modelica description of the model into an executable, *Dymosim*, which performs the simulation (Dynasim 2006). Dymosim is a stand-alone program without any graphical user interface which reads the experiment description from an input file, performs one simulation run, stores the results in Matlab binary format on file, and terminates. Dymosim can be called either from the Dymola's graphic user interface or directly by the user.

Dymola provides, since version 5.0, an interface to Matlab/Simulink for versions above Matlab 5.3 / Simulink 3. Dymola interface to Simulink can be found in Simulink's library browser: *DymolaBlock* block (Dynasim 2006). This block is an interface to the C-code generated by Dymola for the Modelica code. *DymolaBlock* can be connected to other Simulink blocks, and also to other *DymolaBlock* blocks, in the Simulink's workspace window. Simulink synchronizes the numerical 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 (Dynasim 2006). The input variables are supposed to be calculated from other Simulink blocks, while the output variables are calculated from the Modelica model.

2.5 JARA library

JARA is a library of dynamic hybrid models of some fundamental physical-chemical principles (Urquia 2000, Urquia & Dormido 2003). The main application of JARA is the modeling of physical-chemical processes in the context of automatic control. The modeling hypotheses and architecture of JARA are discussed in this section.

JARA was originally written in the "old" Dymola language (Elmqvist et al. 1996). Later on, as a part of this dissertation work (it will be discussed in Chapter 4), the library was translated into Modelica language and adapted for interactive simulation. This new version of the library, called JARA 2i, has been used to compose three of the virtual-labs discussed through this dissertation: control of a chemical reactor, control of an industrial boiler and dynamic behavior of a heat-exchanger. JARA 2i Modelica library can be downloaded from http://www.euclides.dia.uned.es

2.5.1 Fundamental modeling hypotheses of JARA

The usual way of enunciating the mass, energy and momentum balances is by means of the definition of a control volume (CV) (Bird et al. 1975, Incropera & DeWitt 1996, Himmelblau & Bischoff 1992). The properties of the medium inside the control volume are considered time-dependent, but independent of the spatial coordinates. The only exception to this rule is the pressure inside the liquids. The control volumes exchange mass and energy with their environment through certain control planes (CPs). The JARA control volumes and the control planes are considered macroscopic and fixed in the space. All the interactions among control volumes, and all the interactions of a control volume with itself (i.e., chemical reactions inside the control volume), are considered transport phenomena in JARA. This system decomposition into control volumes and transport phenomena suggests that:

1. The control-volume models should contain the equations describing the properties of the medium inside the control volume, as a function of the mass and energy transport through its control planes.

2. The transport-phenomena models should contain equations describing the flow of mass, energy and momentum through the control planes, as a function of the medium properties at these control planes.

Three types of control volumes have been modeled in JARA:

1. Control volume containing a homogeneous solid.

2. Control volume containing an ideal mixture of an arbitrary number of semi-perfect gases.

3. Control volumes containing an ideal, homogeneous liquid mixture, composed of an arbitrary number of components.

The control volumes containing liquid or gaseous mixtures are considered open systems (i.e., they can exchange mass and heat with their environment), and chemical reactions can take place inside them. In both cases, the volume of the control volume is considered a time-dependent property. The control volume containing a solid is considered a closed system (i.e., it only exchanges energy, not mass, with its environment). The only modeled characteristic in solids is the heat conduction (for modeling the walls of reactors, pipes, etc.).

Two kinds of control planes are distinguished in JARA: mass-flow and heat-flow control planes. An arbitrary number of flows can flow through each control plane. The hypotheses made about the properties of the medium inside the control volume determine the nature and number of the control planes:

1. A solid control volume contains only one heat-flow control plane with the following considerations: (i) the solid properties are spatially homogeneous, so that all control planes are equivalent; and (ii) the solid control volume is a closed system, so it has no mass-flow control planes.

2. A gaseous control volume contains one heat-flow control plane and one mass-flow control plane: all the gaseous mixture properties are spatially homogeneous.

3. A liquid control volume has two mass-flow control planes and one heat-flow control plane: (i) as the liquid properties related with the heat-flow are spatially homogeneous, all the heat flow control-planes are equivalent; (ii) as the liquid pressure depends on the position, the simplest and most general control-plane selection is placing a control plane at the control-volume bottom and the other at the control-volume top. Any arbitrarily complex configuration can be modeled by decomposition into this kind of control volume (Urquia 2000).

The JARA models of transport phenomena can be divided into two main groups: (1) mass transport due to pressure or concentration gradients, gravitational acceleration, chemical reactions, liquid-vapor phase changes, etc.; and (2) heat transport due to temperature gradients. The interface variables are grouped into connectors according to this criterion, so that they describe the transport of mass and heat independently.

A hypothesis related to the stirred-mixture approximation is to assume that the fluid going out from a control volume has the same properties that the fluid contained in it. In JARA, this approximation is applied to the calculus of: (1) the temperature and the composition of the fluid leaving a control volume by convection; and (2) the temperature of each mixture component leaving the control volume by diffusion. All the JARA models of transport phenomena make the flow direction reversible during the simulation run. As a consequence, the properties of the flow established between two control planes are calculated from the appropriate control plane at any time.

An important property associated to the transport phenomena is the transport delay. There are different ways of modeling delays in one-dimensional geometry systems (EPRI 1984). The way used in JARA is the "energy balance method" (Incropera & DeWitt 1996). It consists in dividing the flow path

into multiple control volumes. Adjacent control-volumes are connected by the transport-phenomena models describing the heat and mass transport between them. As the number of control volumes increases, the solution gets closer to a transport delay (EPRI 1984).

2.5.2 JARA architecture

The JARA library has been organized in order to facilitate their use and maintenance. The modeling details and the library design rules can be found in (Urquia 2000, Urquia & Dormido 2003). The package hierarchy is shown in Figure 2.4a. JARA is composed of seven packages (see Figures 2.4b – 2.4h):

- The connectors are defined in the *JARA.cuts* package (see Figure 2.4b), and the model interfaces in *JARA.interf* package (see Figure 2.4c).
- The *JARA.gas* package (see Figure 2.4g) gathers models of control volumes containing gaseous mixtures, of gas transport (i.e., gas-flow by convection and diffusion, valves, pumps, etc.) and boundary conditions (i.e., gas-flow and pressure sources).
- Similarly, *JARA.liq* package (see Figure 2.4e) contains the equivalent models for liquid mixtures. The mixtures of liquids and gases are considered ideal and they can be composed of an arbitrary number of components.
- The models related with the heat transport are collected in the *JARA.heat* package (see Figure 2.4d): models of control volumes containing solids, thermal resistances and boundary conditions (heat and temperature sources).
- The *JARA.phase* package (see Figure 2.4f) contains models of vapor-liquid phase-change: boiling and condensation.
- The models of chemical reactions are in *JARA.chReac* package (see Figure 2.4h).

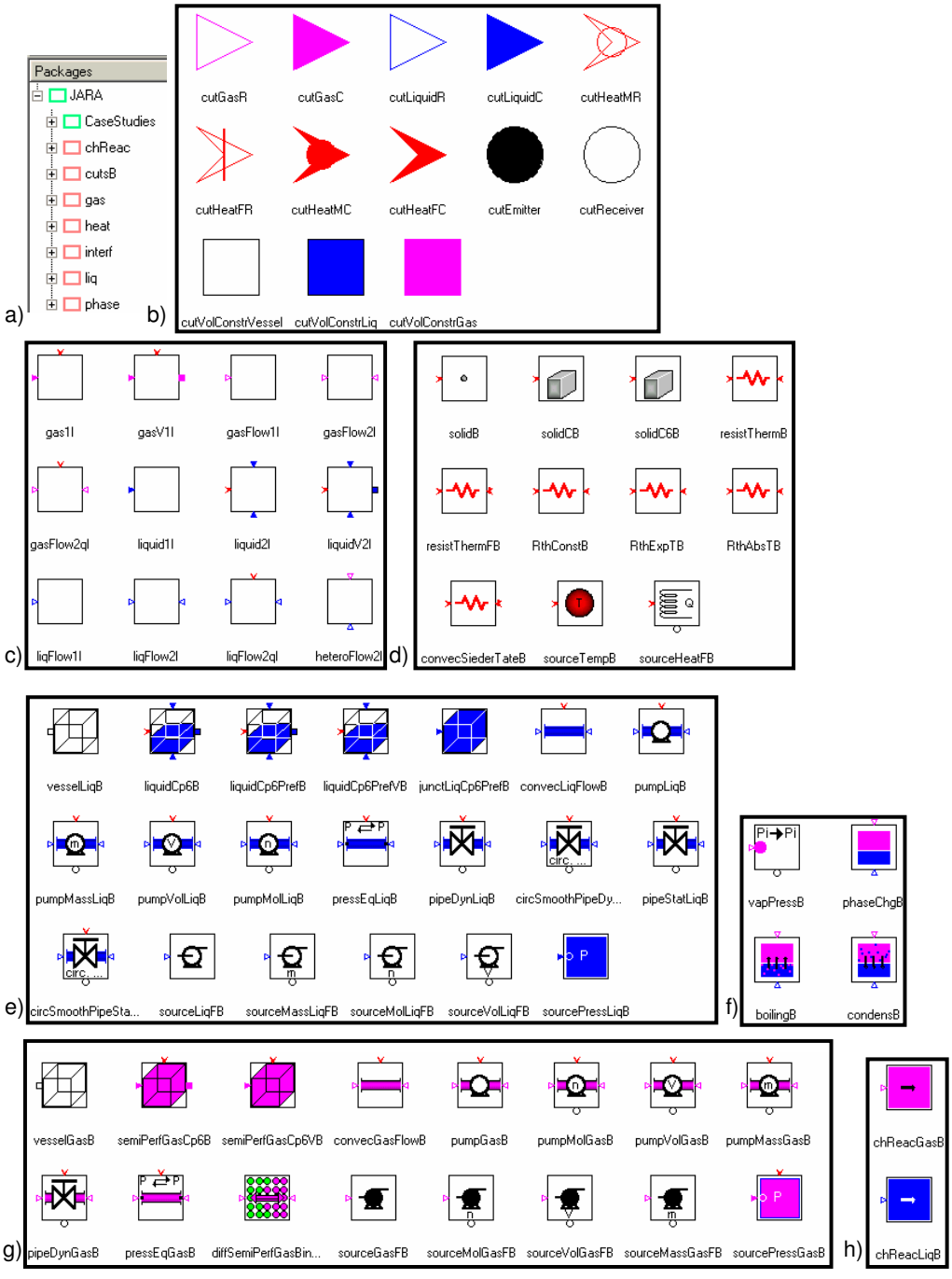


Figure 2.4: a) JARA packages; b) JARA.cuts package; c) JARA.interf package; d) JARA.heat package; e) JARA.liq package; f) JARA.phase package; g) JARA.gas package; h) JARA.chReac package.

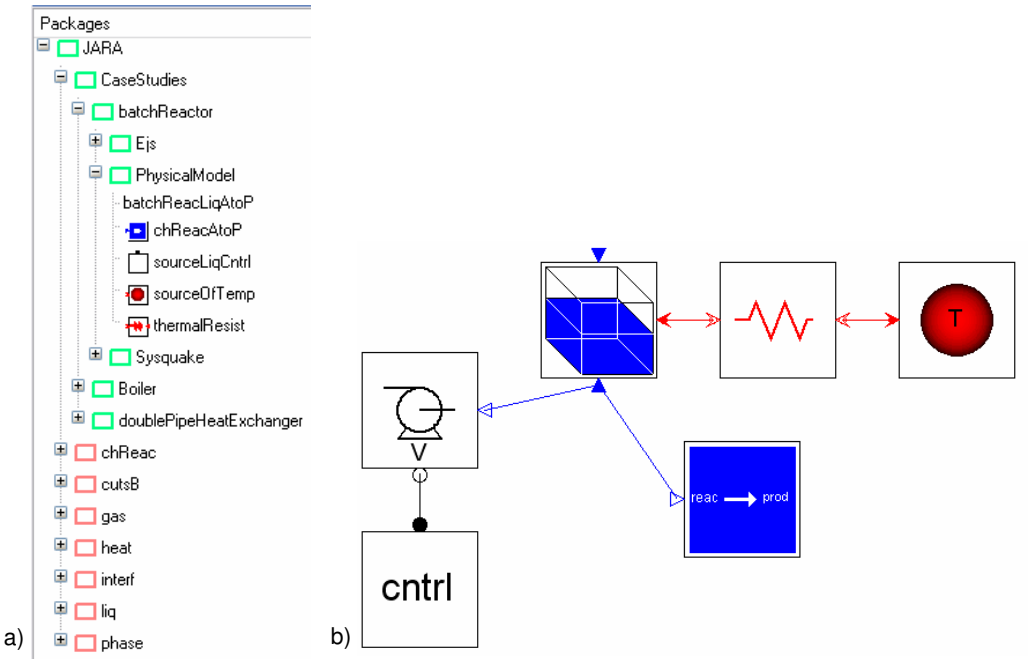


Figure 2.5: a) JARA 2i packages; and b) Modelica diagram of the batch reactor model.

2.5.3 Model of a chemical reactor

The batch reactor model developed by (Urquia 2000), which is based on the mathematical model described in (Froment & Bischoff 1979), has been translated into Modelica language and included in the JARA 2i library (see *batchReactor.PhysicalModel* package in Figure 2.5a).

In a batch reactor having a volume V, an exothermic reaction $A \rightarrow P$ is carried out in the liquid phase. The reaction velocity is $r_A = kC_A$, where k depends on the temperature in the following form: $k = k_{01}exp(-\frac{k_{02}}{T})$, expressed in units of second to minus one. The reactor contains a heat exchanger with a surface A and it can be operated in the following two ways:

1. Using steam with a heat transfer coefficient h_{Ts} at a T_s temperature as heating system.

2. Using cooling water with a heat transfer coefficient h_{Tw} at a T_w temperature as refrigerator.

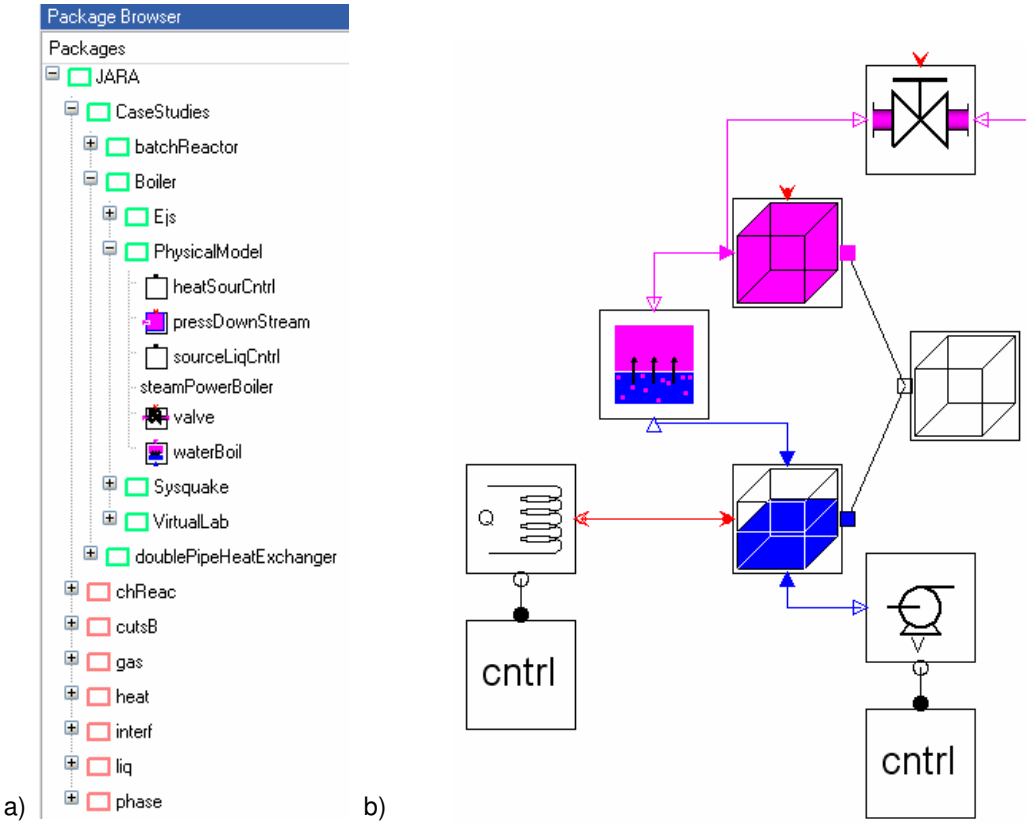


Figure 2.6: a) JARA 2i packages; and b) Modelica diagram of the boiler model.

The Modelica diagram of the chemical reactor model is shown in Figure 2.5b. The model is composed of a CV containing the liquid stored in the reactor, a TP modeling the reaction inside the reactor, a pump model and the model of the heat exchanger. The heat exchanger model is composed of a temperature source and a resistor.

2.5.4 Model of an industrial boiler

The mathematical model of the boiler is found in (Ramirez 1989), and the object-oriented model written in the "old" Dymola language in (Urquia 2000). It has been re-written using JARA 2i components and it has been included in the JARA 2i library (see *Boiler.PhysicalModel* package in Figure 2.6a).

The input of liquid water is placed at the boiler bottom, and the vapor output valve is placed at the top. The output valve has the following constitutive equation: $F^m = (F^0) * \sqrt{(p(p - p_0))}$, where p_0 is the valve output pressure. The

water contained in the boiler is continually heated. The diagram of the boiler model is shown in Figure 2.6b. Two control volumes are considered:

1. A control volume containing the liquid water stored in the boiler.

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 the boiler, the pressure at the valve output and the water pump are modeled using JARA source models.

2.5.5 Model of a double-pipe heat exchanger

The model of the heat exchanger described in (Cutlip & Shacham 1999, Urquia 2000) has been re-written using JARA 2i components and it has been included in the JARA 2i library (see *doublePipeHeatExchanger.PhysicalModel* package in Figure 2.7a). The model diagram is shown in Figure 2.7b.

A mixture of carbon dioxide and sulfur dioxide is cooled by water in a double-pipe heat exchanger (Cutlip & Shacham 1999, Urquia 2000) of length L. The thermic dynamics of the gas mixture, the water and the wall of the inner pipe are considered in the model. The following heat flows have been modeled: the convective heat flow between the gas mixture and the inner wall of the inner pipe, the convective heat flow between the wall of the inner pipe and the water and, finally, the conduction heat flow along the wall of the inner pipe.

The heat exchanger has been divided into $N = 10$ elements to study the dependence of the temperature on the axial coordinate. The length of the elements located at the pipe end, $\frac{L}{2(N-1)}$, is the half of the length of the inner elements. It is assumed that the gas mixture contained in the elements has a uniform temperature. The same assumption has been made related to the temperature of the water and the wall of the inner pipe.

The gas and liquid flow is modeled by pumps that make to flow the established quantity of matter per unit time between the elements. Two modes of operation

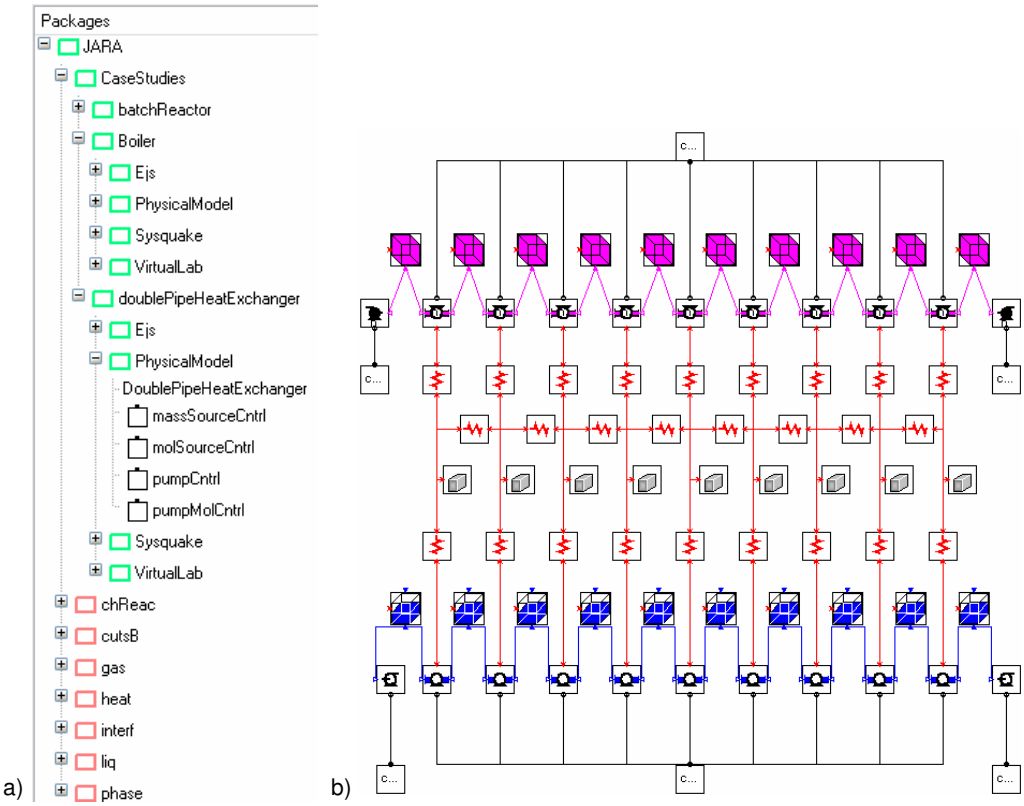


Figure 2.7: a) JARA 2i packages; and b) Modelica diagram of the heat-exchanger model.

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 (Cutlip & Shacham 1999). The center heat exchanger tube is made of copper with a constant thermal conductivity, and the exterior of the steel pipe shell is supposed to be very well insulated.

2.6 Virtual-labs for control engineering education

A virtual-lab is a distributed environment of simulation and animation tools, aimed to perform the interactive simulation of a mathematical model. Two types of interactivity can be distinguished:

– *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, immediately perceiving how these changes

affect the model behavior. An arbitrary number of actions can be made on the model along a 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.

Virtual-labs provide a flexible and user-friendly way to define the experiments to be performed on the model (Jimoyiannis & Komis 2001). In particular, interactive virtual-labs are effective educational resources, well suited for web-based and distance education (Dormido 2004). Due to the special features of the automatic control discipline, control education can be strongly benefited by the use of interactive tools (Navaratna et al. 2001). Some relevant virtual-labs for control education can be found in (Bodson 2003, Muñoz-Gómez et al. 2003, Diaz et al. 2005, Guzman et al. 2005, Erenturk 2005, Ugalde-Loo 2005, Mazaeda et al. 2006).

Automatic control is a multi-faceted field. A good control engineer should master a wide range of topics (Johansson et al. 1998, Wittenmark et al. 1998):

– To have a good understanding of dynamical systems and to know how to describe them.

– To know how different representations of a system (i.e., equations, time responses, frequency responses) are related.

– To master control concepts such as feedback, stability, controllability, observability and to develop an intuition about them.

– To know the interplay between process design and control design. The process design influences strongly the control design. A good process design may avoid processes intrinsically difficult to control.

This wide range of topics makes control education a difficult task. Virtual-labs could be a perfect complement to the traditional labs and lectures. They can

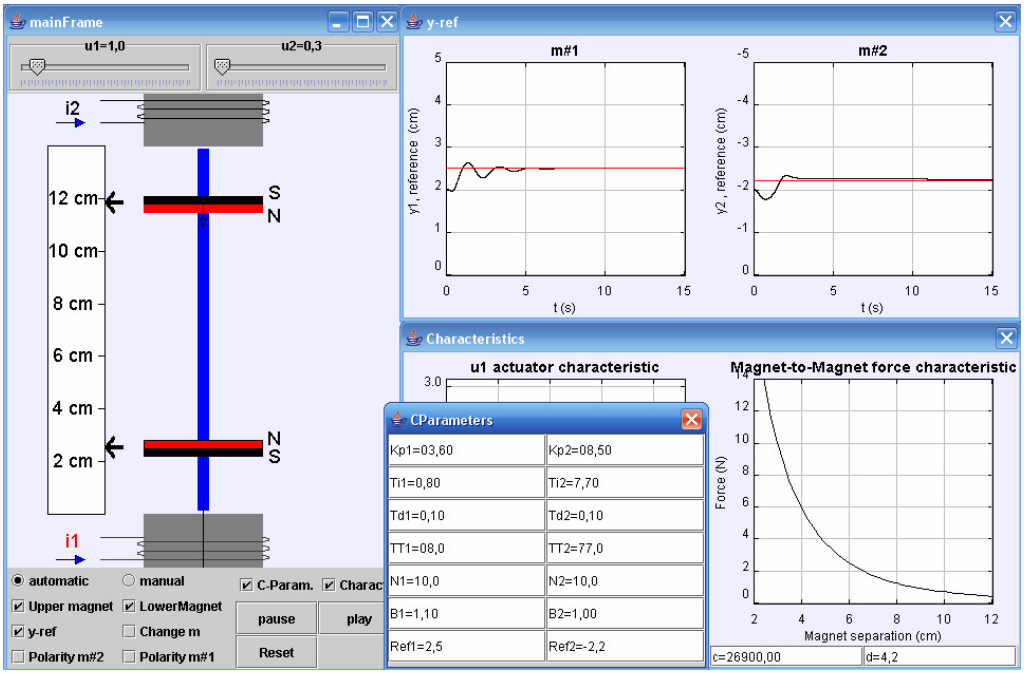


Figure 2.8: View of the magnetic levitator virtual-lab.

be considered half-way between regular labs and lectures. The main idea is to have on the computer screen a multiple-view representation of a given dynamic system, and some of its attributes. These views can then be manipulated directly while keeping the coherence of the representation (Dormido 2004).

Virtual-labs can be used to explain basic concepts, to provide new perspectives of a problem, and to illustrate analysis and design topics. An example of a virtual-lab for control education implemented using Ejs (*EJS* 2007) is shown in Figure 2.8 (Dormido et al. 2004). This virtual-lab illustrates the behavior of a magnetic levitation system. The virtual-lab graphic interface shows the physical system as realistically as possible. Additionally, it shows diagrams and plots of some relevant variables. It is possible, by manipulating the graphic interface, to change the magnets position, the system configuration, the control strategy (manual o decentralized PID) and the parameters of the two PID controllers.

Several software packages for the interactive learning of automatic control have been developed (Dormido et al. 2002, Sanchez et al. 2002). Two of them are ICTools and CCSDemo, from the Automatic Control Department of the Lund Institute of Technology (Johansson et al. 1998, Wittenmark et al. 1998).

Control Station (Cooper & Fina 1999, Cooper & Dougherty 2000, Cooper et al. 2003), developed at the Department of Chemical Engineering of the University of Connecticut, constitutes another good example.

2.7 Interactive simulation tools

The main goal of the interactive simulation tools is to facilitate the virtual-lab implementation, allowing the lab developer to focus on the concepts to be illustrated by the virtual-lab, rather than on programming tasks. Next, some relevant features of the four following interactive simulation tools are discussed: LabVIEW, Sysquake, Ejs and OOCSMP.

2.7.1 LabVIEW

LabVIEW (Laboratory Virtual Instrumentation Engineering) from National Instruments is a graphical development environment for creating flexible and scalable design, control, and test applications (*LabVIEW* 2007). The LabVIEW graphical language, named G, is a dataflow language and cannot be re-interpreted into a text based language. Currently, there is no alternative program that can implement any portion of G code. G language, since version 8.2, has object oriented features.

LabVIEW programs are called virtual instruments (VIs). Each VI has three components: a block diagram, a front panel and a connector panel. Many libraries with functions for data acquisition, signal generation, mathematics, statistics, signal conditioning, analysis and numerous graphical interface elements are provided in several LabVIEW package options.

LabVIEW can be used to build virtual-labs. Examples can be found in (Kostic 2000, Laterburg 2001)

2.7.2 Sysquake

Sysquake is a commercial tool developed at the Federal Institute of Technology in Lausanne (EPFL) by Yves Piguet (Sysquake 2004, Piguet et al. 1999). Sysquake is a Matlab-like program that has strong support for interactive graphics. It is based on LME, an interpreter specialized for numerical computation. LME is widely 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. LME provides the following capacities for modeling systems:

- *lti library.* This library provides methods to create, combine and analyze time-invariant dynamical systems (LTI systems). The LTI system can be defined in three different ways: as a state space model, as a matrix or as a transfer function.

- *ODE solvers.* Sysquake contains the following two ODE solvers: *ode23* and *ode45*. Both ODE solvers are based on a Runge-Kutta algorithm with adaptative time step.

A Sysquake application typically contains several interactive graphical objects, which are displayed simultaneously. Additionally, it can include documentation in form of HTML pages. The 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 directly managed 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 developed with standard compilers.

There are several interactive tools developed with Sysquake (Dimmler & Piguet 2000, Dormido et al. 2002, Diaz et al. 2005, Guzman et al. 2005, Longchamp 2006, Piguet & Longchamp 2006, Guzman et al. 2006). Some applications built by Sysquake users can be downloaded from (*Sysquake* 2007).

2.7.3 Easy Java Simulations

Easy Java Simulations (Ejs) is an open source, Java-based software tool intended to implement virtual-labs (*EJS* 2007, Esquembre 2004). Ejs has been designed to be used by students, under the supervision of educators with a low programming level (Martin et al. 2005d). As a consequence, simplicity was a requirement.

Ejs is based on an original simplification of the "model-view-control" paradigm, structuring the virtual lab in three parts: *introduction*, *model* and *view*.

- Ejs supports including an *introductory part*, composed of HTML pages, in the virtual lab. This introduction is intended to provide information about the simulation and instructions explaining how to use the virtual lab. This feature is important for pedagogical reasons.
- The *model* is the mathematical model describing the system behavior.
- 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.

Ejs guides the user during the model definition process, and it includes a set of ready-to-use visual elements intended to facilitate the virtual-lab view implementation. Ejs automatically performs all the tasks required to generate the virtual lab (i.e., generates the Java source code of the virtual-lab program, compiles the program and packs the resulting object files into a compressed file), which can be run as a stand-alone Java application or as an applet within an HTML page. The user then can readily run the virtual-lab and/or publish it on the Internet.

Ejs includes ODE solvers and algorithms for event detection. Ejs version 3.3 (release 2004) provides a Ejs to Matlab/Simulink interface (Sanchez et al. 2005a,b). 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.

A description of how to use Ejs with Matlab and Simulink can be found in (*EJS* 2007). 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.

2.7.4 Object-Oriented Continuous Modeling Program

Object-Oriented Continuous Modeling Program (OOCSMP) is a continuous simulation language conceived in 1997 as an object-oriented extension to the standard CSMP (Lara & Alfonseca 2003). OOCSMP language is causal and can handle discrete events. A beta version of the compiler and the libraries for Java can be freely downloaded (*OOCSMP* 2007). C-OOL is the compiler for OOCSMP and it is able to generate three different object languages from the OOCSMP

models: plain C++, C++/Amulet and Java. C-OOL automatically generates a user interface that allows the user to control the simulation execution and change the value of object parameters, global variables and simulation parameters.

2.8 Interactive simulation using Modelica

Some efforts have been carried out by other authors in order to provide Modelica with visualization and interactive simulation capabilities. MODIC (Modelica Interactive Control Interface) has been developed for this purpose (Engelson 2000). MODIC allows the user to input and output values via a graphical user interface (Tcl-Tk based) during the simulation. The interface for input values allows the user to change the value of input variables during simulation. From the Modelica side, the communication is performed by using external function calls. These external functions create or modify graphical windows, output values to these windows, or read the value of the input variable currently set by the user.

2.9 Conclusions

The background for this dissertation has been examined in this chapter. An overview of continuous-time modeling and simulation in the context of automatic control has been presented. Some features of object-oriented modeling languages have been discussed, with special emphasis in the Modelica language. In relation to interactive simulation, the concept of virtual-lab and its role in control education has been described. Finally, the capabilities of four interactive simulation tools (i.e., LabVIEW, Sysquake, Ejs and OOCSMP) have been discussed.

3

Batch Interactive Simulation, by Combining the Use of Sysquake and Modelica/Dymola

3.1 Introduction

A novel approach to the implementation of virtual-labs supporting batch interactivity is proposed and it is illustrated by means of four 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 Sysquake.

This approach allows taking advantage of the best features of each tool. Modelica capability for physical modeling, Dymola capability for simulating hybrid-DAE models, and Sysquake capability for:

- Building interactive user interfaces composed of graphical elements (i.e., sliders, menus, Nichols diagrams, time and frequency plots, etc.), whose properties can be linked to the model variables.
- Synthesizing control systems and analyzing linear time-invariant systems.

In order to implement this approach, a Sysquake to Dymosim interface has been programmed. It consists in a set of functions in LME language which can be called from the Sysquake applications. These functions can be downloaded from `http://www.euclides.dia.uned.es`

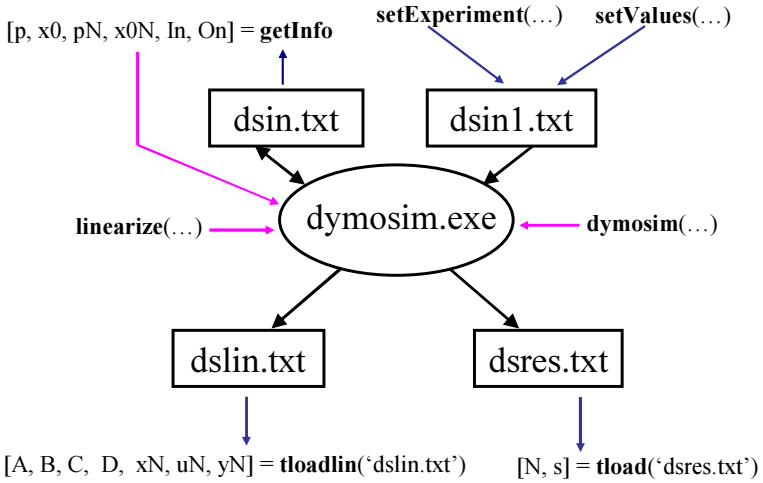


Figure 3.1: Sysquake-Dymosim interface functions.

3.2 Sysquake to Dymosim interface

A Sysquake to Dymosim interface has been implemented. Dymosim (**Dy**namic **mo**del **sim**ulator) is the executable generated by Dymola in order to simulate the model, and then used to perform simulations and initial value computations. It contains the code necessary for continuous simulating and event handling.

The above mentioned interface consists of a set of functions written in LME, which are gathered in a library named *sysquakeDymosimInterface*. These functions synchronize the execution of the *dymosim.exe* file and the Sysquake application. They perform the following tasks (see Figure 3.1, and Appendix A for further details):

- *setExperiment* and *setValues* functions write the experiment description to a text file. This text file can be used as input file for Dymosim.
- *dymosim* and *linearize* functions execute the *dymosim.exe* file in order to simulate and linearize the Modelica model, respectively.
- *tload* and *tloadlin* functions perform the following two operations. Firstly, reading the output file generated by *dymosim.exe* after a model simulation or linearization, respectively. Finally, saving these results to the Sysquake workspace, which then can be used by Sysquake applications.

3.3 Case study I: hysteresis-based controller

The control loop shown in Figure 3.2 is considered. The constitutive relation of the hysteresis-based controller is shown in Figure 3.3. 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. The *view* of the virtual-lab is shown in Figure 3.4:

- The virtual-lab documentation (a set of HTML pages, see Figure 3.5) can be displayed by pressing the "info" icon.
- The transfer function of the plant can be inserted by writing its numerator and denominator in a dialog window. This window is displayed by clicking on the "*System*" item of the "*Settings*" menu.
- A new simulation run can be started by clicking on the "*Run*" item, which is placed on the "*Settings*" menu.

The virtual-lab view is composed of four graphics (see Figure 3.4). Three of them are interactive:

- *"Constitutive relation"* plot: the position of the $\{a, b, c, d, e, f\}$ points of the controller constitutive relation can be changed by dragging the mouse.
- *"Roots"* plot: the plant's zeros and poles can be changed by clicking on the circles and crosses and by dragging the mouse.
- *"Reference"* plot: 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 by dragging the mouse.

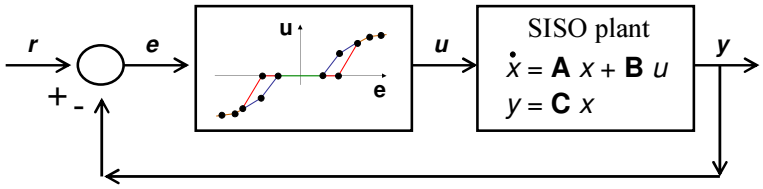


Figure 3.2: Control loop.

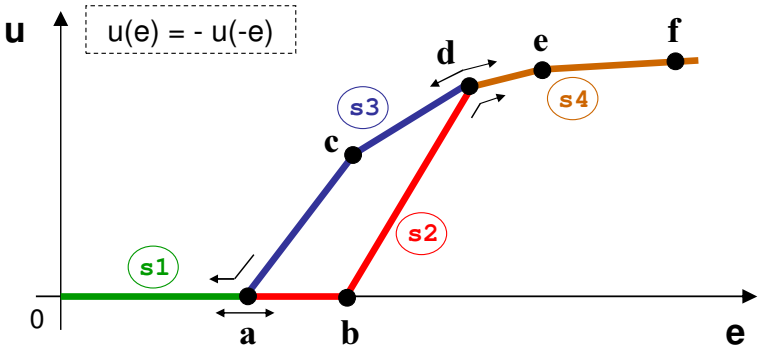


Figure 3.3: Constitutive relation of the controller.

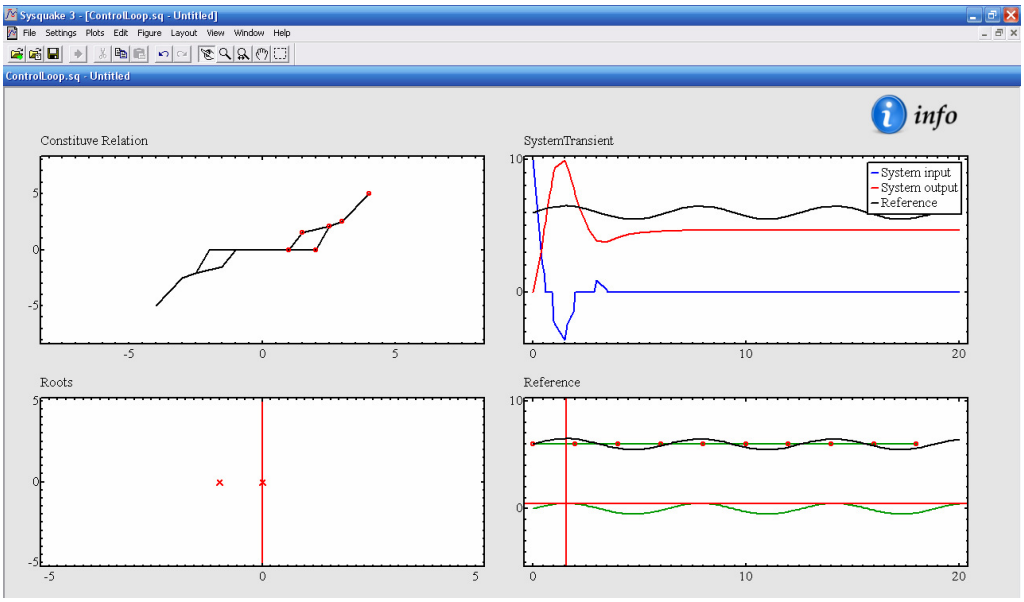


Figure 3.4: View of the control loop virtual-lab.

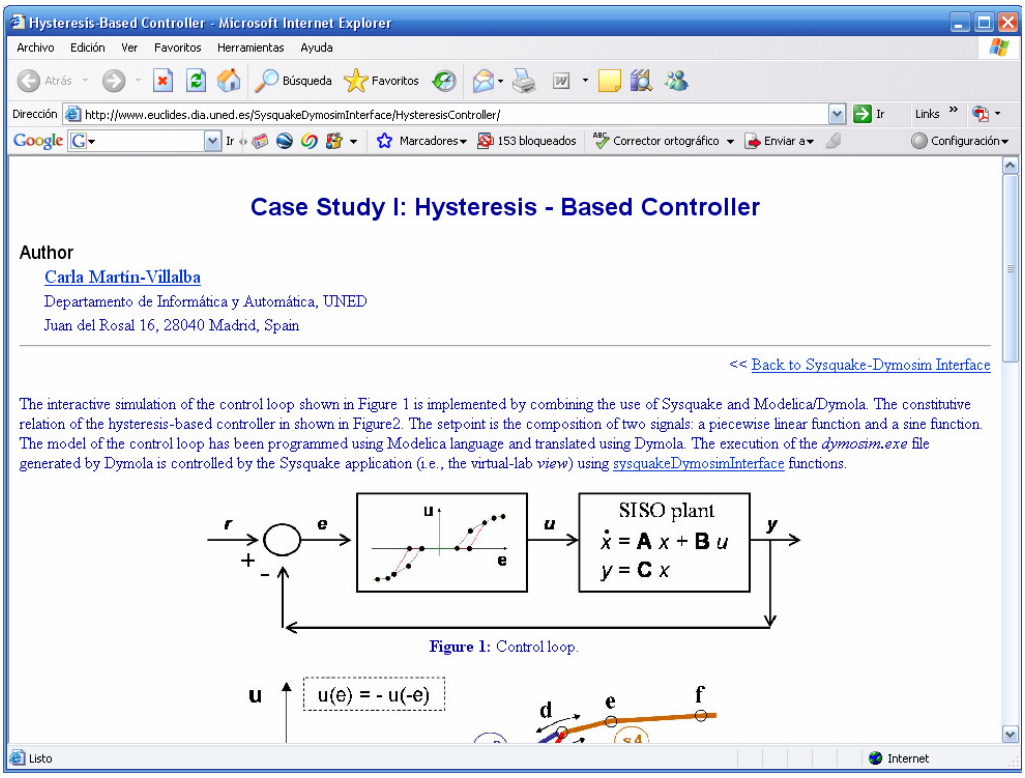


Figure 3.5: Documentation of the control loop virtual-lab.

3.4 Case study II: control of a chemical reactor

The model of a batch chemical reactor has been composed using JARA 2i Modelica library. The diagram of the reactor model is shown in Figure 3.6a. An exothermic reaction $A \rightarrow P$ is carried out in the liquid phase. The reactor contains a heat exchanger, which can be operated with steam and with cooling water. The plant model was described in Section 2.5.3.

The diagram of the Modelica model describing the closed-loop system is shown in Figure 3.6b. It has been used the PID controller model included in the standard Modelica library (*Modelica* 2007), which has been designed according to the model described in (Åström & Hagglund 1995). This model has limited output, anti-windup compensation and setpoint weightings. It has the following parameters:

K_p	Proportional gain.
T_i	Integral time constant.
T_d	Derivative time constant.

w_p	Setpoint weight for the proportional term.
w_d	Setpoint weight for the derivative term.
N_i	Anti-windup compensator constant.
N_d	Derivative filter parameter.
y_{min}	Lower limit for the output.
y_{max}	Upper limit for the output.

The reactor's operation policy is the following (Froment & Bischoff 1979):

1. Fill up the reactor with the reacting liquid (the inflow is controlled by a PID).
2. Preheat to certain temperature (T_1), and let the reaction proceed adiabatically.
3. Start cooling when either the maximum allowable reaction temperature (T_{max}) occurs or the desired conversion is reached (x_d), and cool down to the desired temperature (T_d).
4. Empty the reactor.

The virtual-lab view is shown in Figure 3.7. It contains sliders to change the model parameters, the initial value of the state variables and the input variables. The *"Settings"* menu allows the user to (see Figure 3.7):

1. Change the parameters of the control policy (i.e., T_1, T_{max}, x_d, T_d and PID parameters).
2. Set the communication interval and the total simulation time.
3. Launch a simulation run.

The view contains an "*info*" icon that displays the virtual-lab documentation. Also, it has three plots representing the time-evolution of the relevant process variables (i.e., the mass of A, P and water, the mixture temperature, and the pump throughput).

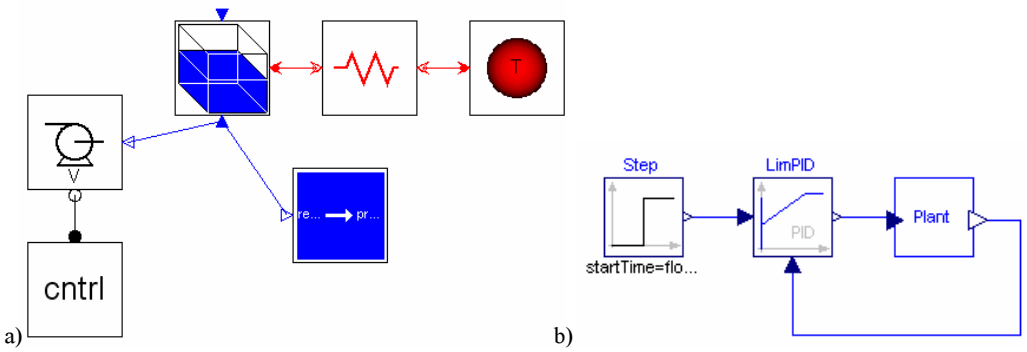


Figure 3.6: Diagram of the reactor Modelica model: a) open-loop system; and b) closed-loop system.

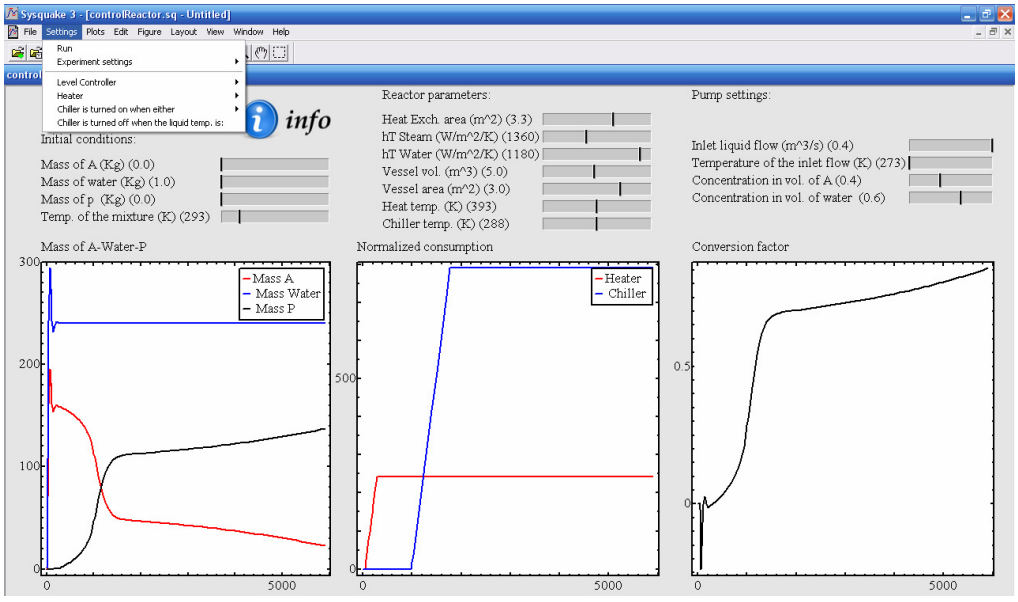


Figure 3.7: View of the chemical reactor virtual-lab.

3.5 Case study III: control of a double-pipe heat exchanger

The JARA 2i model of a double-pipe heat exchanger was discussed in Section 2.5.5. The model diagram is shown in Figure 3.8a. The goal of this virtual-lab is to illustrate the application to the heat exchanger of some linearization and control techniques.

Three different Modelica models has been composed using the JARA 2i library and components from the standard Modelica library:

1. The open-loop system (see Figure 3.8a).
2. The heat-exchanger controlled using a PID (see Figure 3.8b).
3. The heat-exchanger controlled using a compensator (see Figure 3.8c).

In addition, a Sysquake application has been programmed. It implements the virtual-lab view and controls the execution of the three *Dymosim* files: the *Dymosim* file that simulates the open-loop plant, and the two *Dymosim* files that simulate the plant controlled using a PID and a compensator respectively.

The features of this Sysquake application, that constitutes the virtual-lab core, include:

1. The application to the heat-exchanger model of several identification techniques.
2. The design of control strategies (using the linear models previously obtained by applying the identification techniques).

The challenge is to control the gas exit temperature by manipulating the water flow. In addition, the virtual-lab view contains an "*info*" icon that displays the documentation (see Figure 3.9).

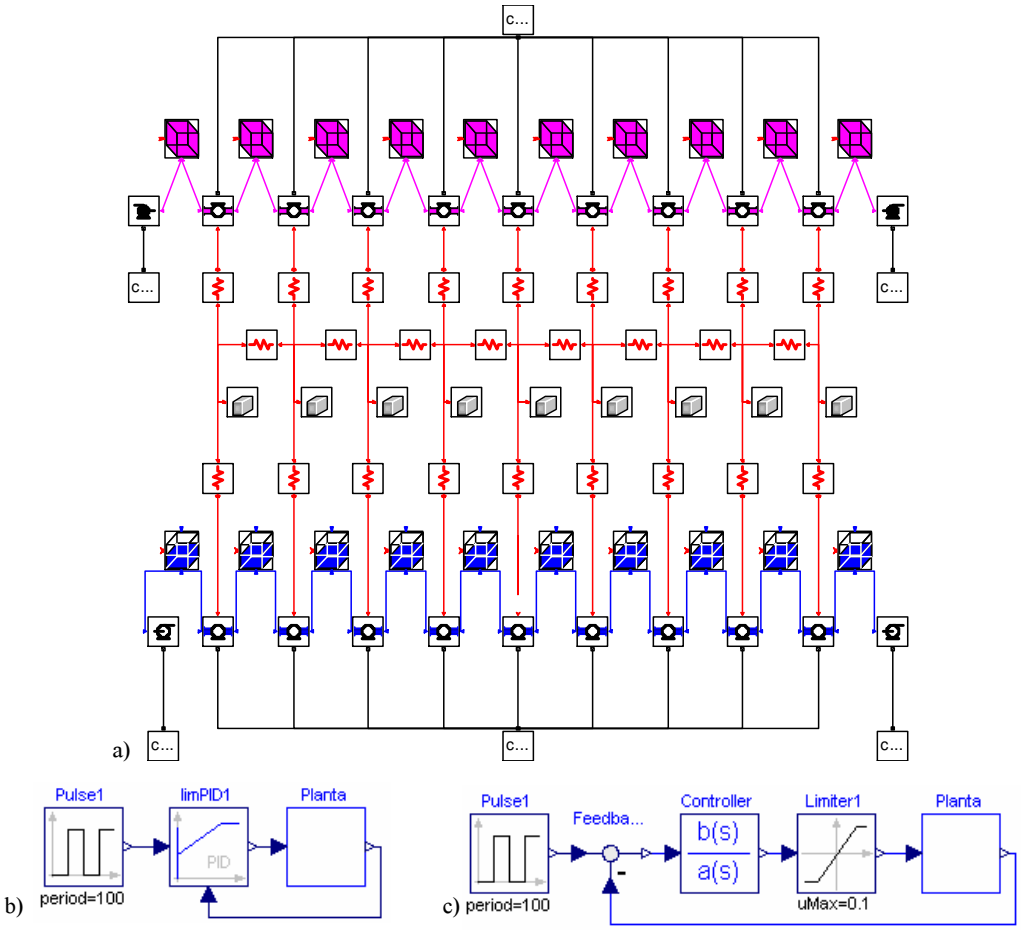


Figure 3.8: Diagram of the heat-exchanger Modelica model: a) open-loop plant; b) plant controlled using a PID; and c) plant controlled using a compensator.

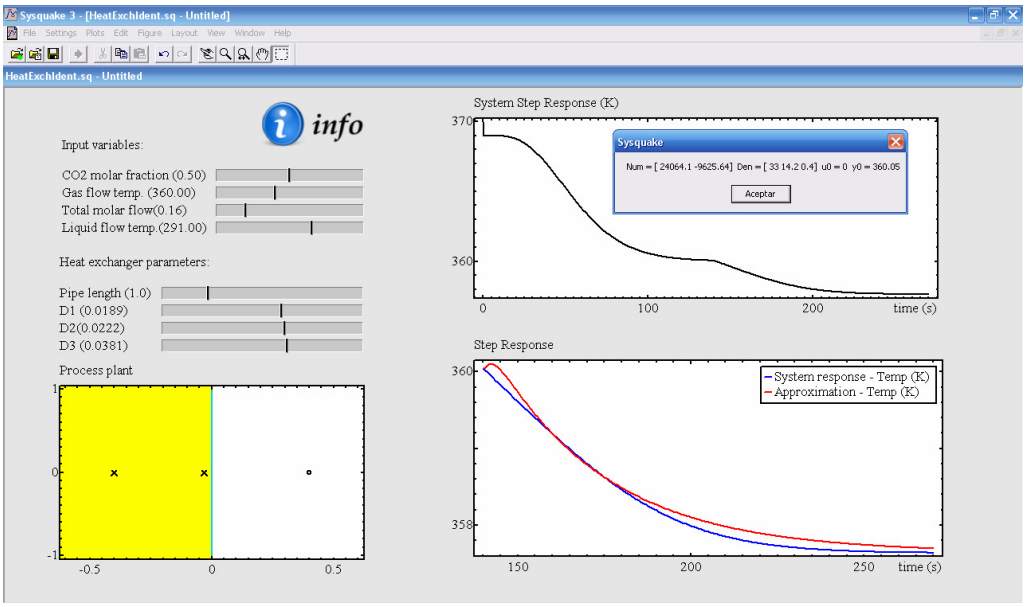
3.5.1 Plant identification

The virtual-lab supports the automatic calculation of the plant linearized model. This calculation is performed as follows (see Figure 3.9a):

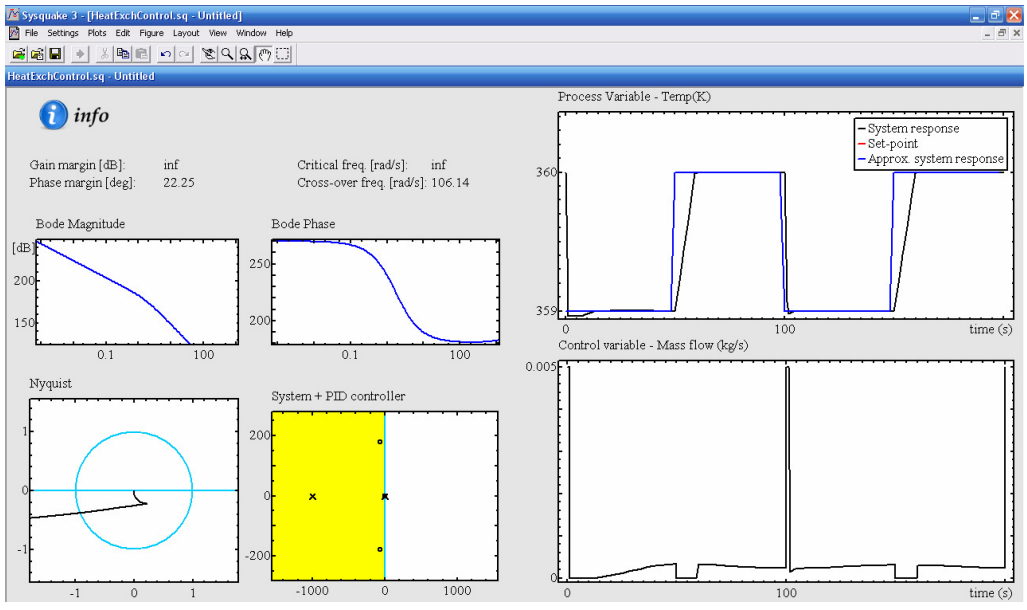
1. The change in the value of the gas exit temperature, in response to a step in the water flow, is calculated simulating the heat exchanger model.
2. A transfer function (abbreviated: TF) is fitted to this response.

During this identification procedure, the virtual-lab user is allowed to:

1. Change the parameter values and the input variable values of the heat exchanger model, the simulation communication interval and the total simulation time.



a)



b)

Figure 3.9: View of the double-pipe heat-exchanger virtual-lab: a) plant linearization; and b) controller synthesis.

2. Choose among different identification methods, including "*first order TF with delay*", "*second order TF with delay*" and "*non-parametric identification*".

3. Modify the obtained TF.

4. Analyze the obtained TF by means of Bode and zero-pole diagrams, and robustness margins.

5. Start the simulation run.

6. Export the calculated TF to another Sysquake application.

3.5.2 Controller synthesis and analysis

In addition, the virtual-lab automates the controller synthesis and analysis. The virtual-lab supports the following user's operations (see Figure 3.9b):

1. To import the TF previously identified.

2. To analyze the TF characteristics using Nyquist, Nichols and Bode diagrams.

3. To choose the controller type. Possible options are: PID, lead and lag compensators.

4. To synthesize the controller (i.e., to set the value of the PID's parameters).

5. To specify the error and the phase margin of the system controlled by the lead or lag compensators.

6. To simulate the closed-loop linear and non-linear models.

3.5.3 Example of use

An experience using the heat-exchanger virtual-lab will be described below. The operation conditions of the heat exchanger are shown in Figure 3.9a. A change in the value of the the water-flow from 0 to 10^{-4} kg/s has been applied at time

150 s to the heat exchanger. A TF has been fitted to the change in the value of the gas exit temperature in response to the step change in the water-flow. A first order identification method, that uses the times to reach 28.3% and 63.2% response, has been applied. The following TF has been obtained:

$$\frac{24064.1s - 10240}{33.3s^2 + 15.17s + 0.43} \tag{3.1}$$

A PID to control the plant has been designed. The TF previously obtained has been used in the design process. The PID controller has the following parameters: $Kp = 0.05$, $T_i = 1$, $T_d = 0.01$, $w_p = 1$, $w_d = 1$, $N_i = 0.9$, $N_d = 10$, $y_{min} = 0$ and $y_{max} = 5 \cdot 10^{-3}$. The evolution of the gas exit temperature tracking the set-point is shown in Figure 3.9b.

3.6 Case study IV: control of an industrial boiler

JARA 2i Modelica library has been used to compose the model of an industrial boiler. that was explained in Section 2.5.4. The input of liquid water is located at the boiler bottom, and the vapor output valve is placed at the boiler top. The water contained inside the boiler is continually heated.

The model diagram is shown in Figure 3.10a. It is composed of two control volumes, in which the mass and energy balances are formulated: (1) a control volume containing the liquid water stored in the boiler; and (2) a control volume containing the generated vapor. The model of the boiling process connects both control volumes. The heat flow from the heater to the water, the pressure at the valve output and the water pump are modeled using JARA source models.

This virtual-lab is intended to illustrate the identification of the industrial boiler and the synthesis of the boiler control system. This control system is composed of two decoupled control loops:

1. The water level inside the boiler is controlled by manipulating the pump throughput.

2. The output flow of vapor is controlled by manipulating the heater power.

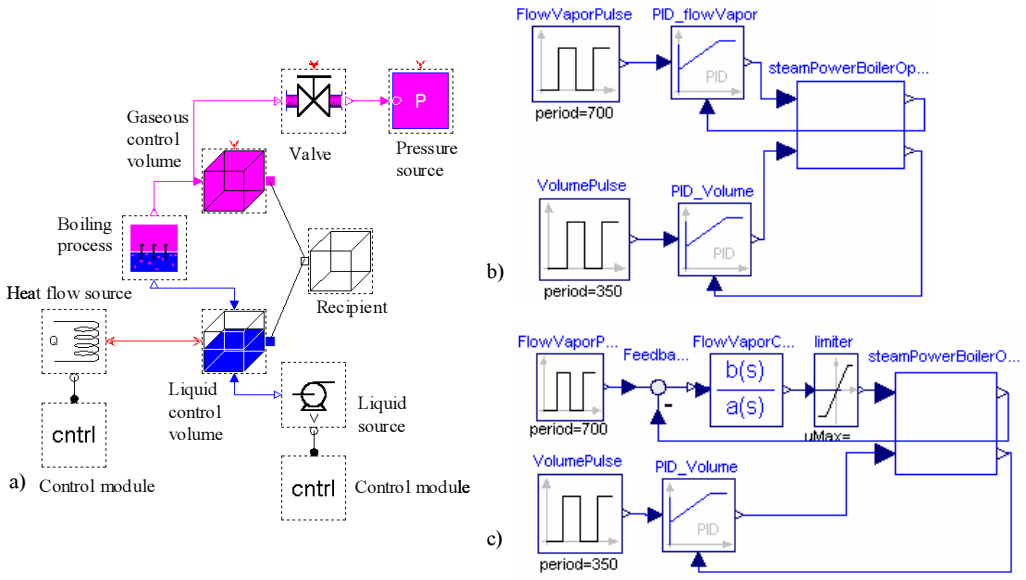


Figure 3.10: Diagram of the boiler Modelica model composed using JARA: a) open-loop plant; b) plant controlled using two PID; and c) plant controlled using a PID to control the water level inside the boiler and a compensator to control the output flow of vapor.

The identification and synthesis procedures are similar to the one discussed in Section 3.5. The virtual-lab view contains an "*info*" icon that displays the documentation.

Three different Modelica models has been built to identify and control the system:

1. The open-loop system (see Figure 3.10a).
2. The boiler controlled using two PIDs (see Figure 3.10b).
3. The boiler controlled using a PID to control the water level inside the boiler and a compensator to control the output flow of vapor (see Figure 3.10c).

The identification and synthesis procedures are briefly described next.

3.6.1 Plant identification

The virtual-lab user is allowed to choose interactively the plant's operation point. This is accomplished by setting the value of:

– The mass and temperature of the liquid and the vapor inside the boiler.

– The valve opening and its downstream pressure.

– The flow and inlet temperature of the water.

Once the operation point has been set, the user can launch the calculation of the two TF: (1) a TF from the "*pump throughput*" (input) to the "*water level*" (output); and (2) a TF from the "*heater power*" (input) to the "*vapor flow*" (output). These TF are automatically fitted to simulated step responses by the virtual-lab. The user can choose among the following identification methods (see Figure 3.11a): "*first order TF with delay*", "*second order TF with delay*" and "*non-parametric identification*".

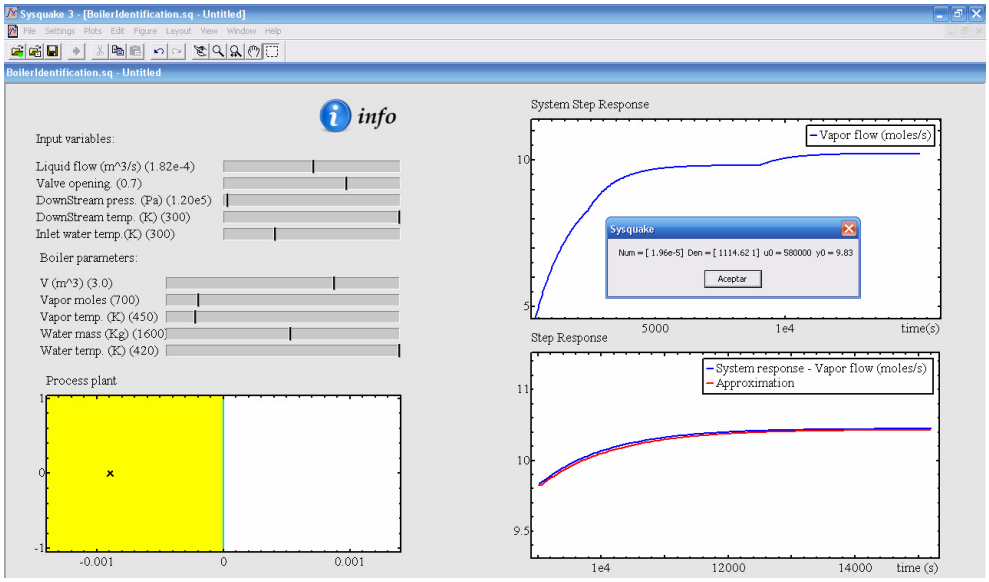
The virtual-lab supports a set of graphical methods to analyze the fitted TF, including Bode and pole-zero diagrams, and it automatically computes the robustness margin. In addition, the virtual-lab allows to export the TF to any other Sysquake application.

3.6.2 Controller synthesis and analysis

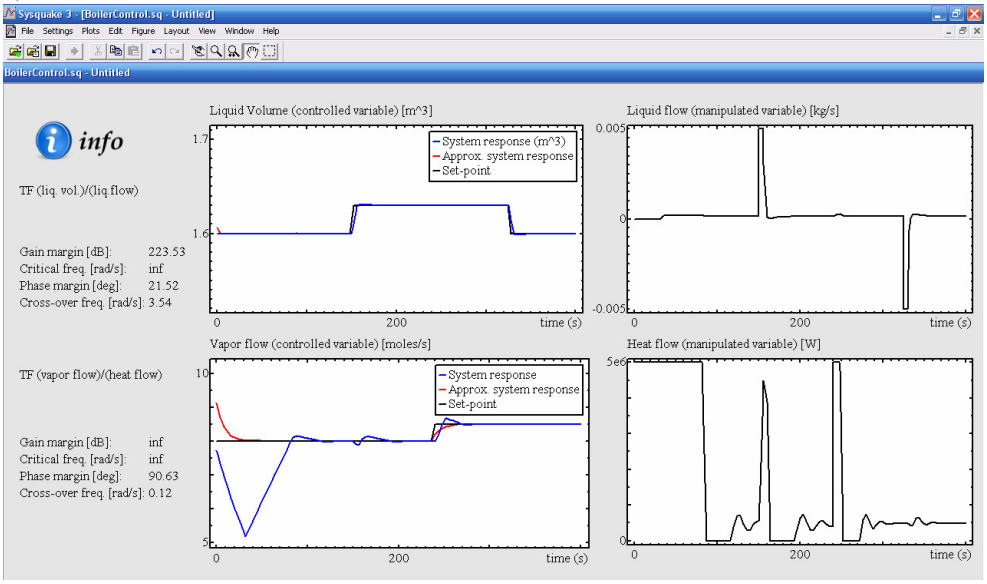
The virtual-lab facilitates the design and analysis of the two controllers (see Figure 3.11b). The water level inside the boiler is controlled using a PID. The gas flow can be controlled using a PID, a lead or a lag compensator. The user can change the controller parameters, and the error and phase-margin specifications of the compensation networks.

3.6.3 Example of use

An experience using the industrial boiler virtual-lab will be described below. The following TF has been considered to describe the changes in the liquid levels due to changes in the pump flow: $\frac{1.3}{s}$. A change in value of the heat flow from $5.8 \cdot 10^5$ to $6 \cdot 10^5$ W has been applied to the heat exchanger at time 9000 s. The operation conditions of the boiler are shown in Figure 3.11a. A TF has been fitted to the vapor flow by applying a first order identification method. The following TF has been obtained:



a)



b)

Figure 3.11: View of the boiler virtual-lab: a) plant linearization; and b) controller synthesis.

$$\frac{1.96 \cdot 10^{-5}}{1114.6s + 1} \tag{3.2}$$

Two PID controllers have been designed. The PID that controls the liquid volume inside the boiler has the following parameters: $Kp = 1$, $T_i = 9$, $T_d = 1 \cdot 10^{-3}$, $w_p = 1$, $w_d = 1$, $N_i = 0.9$, $N_d = 10$, $y_{min} = -0.01$ and $y_{max} = 0.01$. The PID that controls the vapor output flow has the following parameters: $Kp = 7 \cdot 10^6$, $T_i = 1.1$, $T_d = 3 \cdot 10^{-3}$, $w_p = 1$, $w_d = 1$, $N_i = 0.9$, $N_d = 10$, $y_{min} = 0$ and $y_{max} = 5 \cdot 10^6$.

The time evolution of the set-points, the manipulated variables and the control variables are shown in Figure 3.11b.

3.7 Conclusions

The feasibility of combining Modelica/Dymola with Sysquake, for implementing virtual-labs with batch interactivity has been demonstrated. Sysquake is a software tool specifically oriented to develop virtual-labs. The use of Modelica language considerably reduces the modelling effort and facilitates the model reuse.

In order to implement this software combination approach a Sysquake-to-Dymosim interface has been programmed. This approach has been successfully applied to the implementation of virtual-labs intended for control education.

4

Modeling Methodology for *Runtime* Interactive Simulation

4.1 Introduction

Two different approaches for implementing virtual-labs with *runtime* interactivity have been proposed in this dissertation:

Approach A. *Implementing virtual-labs by combining the use of Easy Java Simulations, Matlab/Simulink and Modelica/Dymola.* The virtual-lab model is described using Modelica and the virtual-lab view is implemented using Ejs. The model-view communication is carried out through Matlab/Simulink. This approach will be discussed in Chapter 5.

Approach B. *Describing virtual-labs using only Modelica language.* The virtual-lab model is described using Modelica. The virtual-lab view is composed using *VirtualLabBuilder* Modelica library, which contains Modelica models implementing graphic interactive elements, such as containers, animated geometric shapes and interactive controls. These models allow the virtual-lab developer: (1) to compose the view; and (2) to link the visual properties of the virtual-lab view with the model variables. The components of the library contain the code required to perform the bidirectional communication between the view and the model. In addition, *VirtualLabBuilder* library supports including documentation (HTML pages) in the virtual-lab. The

design and programming of *VirtualLabBuilder* is part of the research work presented in this dissertation. It will be discussed in Chapters 6 and 7.

In both approaches, the virtual-lab model is described using the Modelica language. A systematic methodology is proposed in this dissertation for adapting any Modelica model into a description suitable for *runtime* interactive simulation. The model modifications required for Approach A and B are slightly different, due basically to the following two facts:

1. The causality of the Modelica model interface needs to be explicitly set in Approach A. The reason is that, in Approach A, the Modelica model needs to be embedded within a Simulink block of *DymolaBlock* type.

2. The code required to implement the user's changes in the value of the interactive quantities is pre-defined in some components of the *VirtualLabBuilder* Modelica library. Therefore, this code does not need to be included in the virtual-lab model description for Approach B .

The model modification methodologies for Approaches A and B are discussed in this chapter, and they are applied for adapting JARA Modelica library to interactive simulation. The adapted library, called JARA 2i, has been used to compose three of the virtual-labs discussed in Chapters 5 and 6: control of a chemical reactor, control of an industrial boiler and dynamic behavior of a heat-exchanger. Finally, support to multiple selections of the model state variables will be discussed in this chapter and illustrated by means of a case study.

4.2 Model description for interactive simulation

A methodology for transforming any Modelica model into a description suitable for interactive simulation is proposed in this section. The following terminology will be used. The original model of the system is called *physical model*, and its reformulation for interactive simulation is called *interactive model*.

The model shown in Figure 4.1 will be used to illustrate the discussion. The voltage applied to the pump (v) is an *input variable* (i.e., its value is not calculated

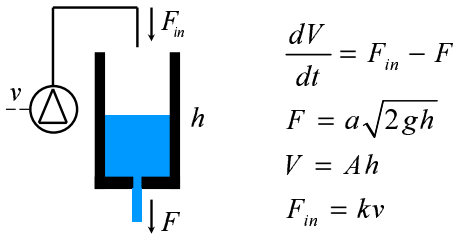


Figure 4.1: Tank model.

from the model equations). The cross-sections of the tank (A) and the outlet hole (a), the pump parameter (k) and the gravitational acceleration (g) are *parameters* (i.e., time-independent quantities of the model). The liquid volume (V), the input and output flows (F_{in}, F), and the liquid level (h) are *time-dependent variables* of the physical model.

The model of the system shown in Figure 4.1 can be described by the connection of following three components:

1. The *pump*, modeling the input flow of liquid ($F_{in} = kv$).
2. The *tank*, describing the conservation of the liquid volume ($dV/dt = F_{in} - F$) and the relationship between the volume and the liquid level ($V = Ah$).
3. The *pipe*, describing the output flow of liquid ($F = a\sqrt{2gh}$).

4.2.1 Interactive quantities

The virtual-lab design process includes selecting the *interactive quantities*. These are the model quantities whose values can be interactively changed by the user during the simulation run. The virtual-lab goal is to illustrate the dependence between the model dynamic behavior and the value of those quantities.

Interactive quantities can be parameters, input variables, and time-dependent variables of the *physical model*. For instance, some interactive quantities of the model shown in Figure 4.1 could be the following:

- *Parameters*: the cross-sections of the tank (A) and the outlet hole (a), and the pump parameter (k).

– *Time-dependent variables*: the liquid level (h).

– *Input variables*: the voltage applied to the pump (v).

The *interactive model* combines the dynamic behavior described in the physical model and the abrupt changes in the value of the interactive quantities produced by the user's actions:

1. The evolution in time of the interactive *time-dependent quantities* is described by the physical model equations. In addition, their value can change abruptly as a result of the user's interaction.

2. The value of the interactive *model parameters* can be abruptly changed by the user's action, remaining constant between consecutive interactive changes.

3. The value of the interactive *input variables* is interactively set by the user. Their value changes abruptly as a result of the user's action, remaining constant between consecutive changes.

Parameters represent time-independent quantities. Input variables represent boundary conditions which are not calculated from the model equations. Although they are conceptually different, the dynamic behavior of interactive parameters and interactive input variables is the same. Their value change abruptly at the interaction instants, remaining constant between consecutive changes. As a consequence, both types of interactive quantities are described in the same manner in the interactive model.

4.2.2 Description of the interactive quantities

In order to support abrupt changes in their values during the simulation run, interactive quantities need to be state variables of the interactive model. The *interactive model* is obtained from the *physical model* by reformulating (when required) the declaration and evaluation of the interactive quantities, so that they become state variables of the interactive model. To this end, the virtual-lab developer has to perform the following tasks.

```
model tank
  parameter Real Ainitial "Initial value of the tank section";
  Real A (start = Ainitial) "Tank section - Interactive quantity";
  ...
equation
  der(A) = 0;
  ...
end tank;
```

Modelica Code 4.1: Tank section (A) redefined as interactive quantity.

– *Time-dependent variables* need to be selected as state variables. Modelica and Dymola support the user's control on the state variables selection, via the *stateSelect* attribute of Real variables (Mattsson et al. 2000, Otter & Olsson 2002, Dynasim 2006, Fritzson 2004). This attribute values include "*never*" (the variable will never be selected as state variable) and "*always*" (the variable will always be used as a state). This feature allows the user to select the model state variables without performing any manipulation on the model equations. The required model manipulations are automatically performed by Dymola.

– *Parameters* and *input variables* are redefined as time-dependent variables with zero time-derivative, and they are selected as state variables. For instance, the parameter A of the tank model shown in Figure 4.1 should be a *Real* variable of the interactive model, calculated from the equation $\mathsf{der(A)} = 0$ (see Modelica Code 4.1).

Let's consider that all the interactive quantities can be simultaneously selected as state variables. The description of interactive models without this restriction will be discussed in Section 4.4. Changes in the interactive quantities are performed as state re-initialization events by using the Modelica's $reinit(x, expr)$ operator. It re-initializes an state variable (x) with the value obtained by evaluating an expression ($expr$), at the event instant. These changes are triggered using *when* clauses.

The required code to implement the user's changes in the value of the interactive quantities (i.e., re-initialization events triggered using *when* clauses)

is pre-defined in the interactive control elements contained in the *VirtualLab-Builder* Modelica library. Therefore, in case of virtual-labs implemented using Modelica/Dymola and the *VirtualLabBuilder* Modelica library, the virtual-lab developer does not need to perform any further modification in the model. On the contrary, in case of virtual-labs implemented by combining Ejs, Matlab/Simulink and Modelica/Dymola, the code to implement the user's changes in the value of the interactive quantities has to be included in the interactive model by the virtual-lab developer.

Defining the interactive parameters and input variables as state variables increases the number of state variables. This has an unwanted effect: it slows down the simulation. We could think of redefining the interactive parameters and input variables as discrete-time variables or, alternatively, as input variables whose values are provided by the virtual-lab view. In this way, the number of state variables would not be increased. However, as it is discussed next, this is not a valid approach. Dymola automatically performs model manipulations in order to formulate the model according to the requested state selection. The problem is that these model manipulations can require differentiating an interactive parameter or input variable, which results in an error being generated. An example is shown next.

Consider the model shown in Figure 4.1. It is formulated according to the state selection $e_1 = \{V\}$. In order for h to be a state variable instead of V, the model can be manipulated as shown below. The variable to be evaluated from each equation is written within square brackets.

$$[F] = a\sqrt{2gh} \tag{4.1}$$

$$[F_{in}] = kv \tag{4.2}$$

$$[derV] = F_{in} - F \tag{4.3}$$

$$[V] = Ah \tag{4.4}$$

$$derV = \dot{A}h + A\left[\dot{h}\right] \tag{4.5}$$

The time-derivative of the tank cross-section (i.e., $\dot{A}$) appears in Eq. (4.5). If the interactive quantity A is defined as an input variable, then an error is produced: Dymola can not differentiate an input variable. The same problem arises if A is defined as a discrete-time variable. A valid approach is the previously discussed: defining the interactive parameters and input variables as constant state variables (i.e., $\dot{A} = 0$). The interactive changes in the value of these quantities are implemented by re-initializing their values.

The physical models have to be modified as was described in this section. In case of the model shown in Figure 4.1, the description of the physical components composing the physical model could be modified as shown below. It is supposed that h is selected as state variable.

The selection of h as state variable is controlled via the *StateSelect* attribute. The interactive parameters (A, a, k) and the input variable (v) have been defined as constant state variables (see Modelica Code 4.2).

```
model tank
  Real h (stateSelect = StateSelect.always) "Liquid level";
  Real V (stateSelect = StateSelect.never) "Liquid volume";
  parameter Real Ainitial "Initial value of the tank section";
  Real A (start = Ainitial) "Tank section - Interactive quantity";
  ...
equation
  der(A) = 0;
  ...
end tank;

model pipe
  Real F (stateSelect = StateSelect.never) "Liquid flow";
  parameter Real aInitial = 1 "Initial value of the pipe section";
  Real a (start = aInitial) "Pipe section - Interactive quantity";
  ...
equation
  der(a) = 0;
  ...
end pipe;

model pump
  parameter Real vInitial "Initial value of the applied voltage";
  Real v (start = vInitial) "Voltage applied to the pump - Interactive";
  parameter Real kInitial "Initial value of the pump parameter";
  Real k (start = kInitial) "Pump parameter - Interactive quantity";
  ...
equation
  der(v) = 0;
  der(k) = 0;
  ...
end pump;
```

Modelica Code 4.2: Tank model with the following interactive quantities: A, a, v, k.

4.3 Design of JARA 2i

JARA library (Urquia 2000, Urquia & Dormido 2003) has been translated into Modelica language and adapted for interactive simulation by applying the methodology proposed in Section 4.2. The new version of the library is called JARA 2i. The library code, its on-line documentation and some examples of use are available at `http://www.euclides.dia.uned.es`

JARA 2i is intended to be used for *batch* and *runtime* interactive simulation. In order to be adapted for *runtime* interactive simulation, the model needs to be modified as described in Section 4.2. These modifications imply the increment of the number of the state variables, with the unwanted effect of slowing down the simulation. On the other hand, no model modifications are required for batch interactive simulation using Sysquake. In consequence, the model modifications have been coded in a way that they can be conditionally included or removed from the models.

Two global Boolean parameters have been defined: `Ejs` and `Sysquake`. These two parameters are declared as *inner* variables to the JARA components and *outer* variables to the physical models. *if-then-else* Modelica clauses are used to include or remove code from the models depending on the value of these two variables. An example is shown in Modelica Code 4.3.

If `Ejs = true` and `Sysquake = false`, then the equation setting the time derivative of the tank cross-section to zero is activated. On the other hand, if `Ejs = false` and `Sysquake = true`, then the tank cross-section is calculated from `A = Ainitial`. The `Ainitial` parameter is the initial value of `A`.

```
model tank
  inner Boolean Ejs;
  inner Boolean Sysquake;
  parameter Real Aintial = 1 "Initial value of the tank section";
  Real h (stateSelect = StateSelect.always) "Liquid level";
  Real V (stateSelect = StateSelect.never) "Liquid volume";
  parameter Real Ainitial "Initial value of the tank section";
  Real A (start = Ainitial) "Tank section - Interactive quantity";
  ...
equation
  if Ejs then
    der(A) = 0;
  end if;
  if Sysquake then
    A = Ainitial;
  end if;
  ...
end tank;
```

Modelica Code 4.3: Tank model adapted for interactive simulation using Ejs and Sysquake.

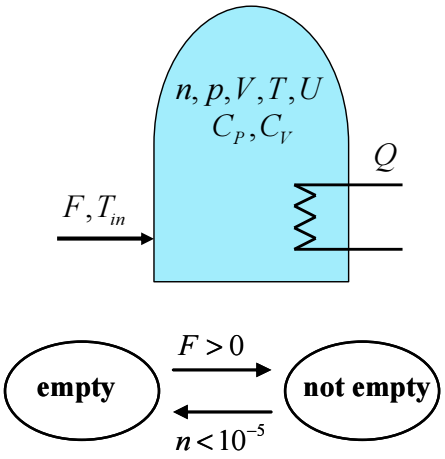
4.4 Supporting several selections of the state variables

As it was discussed in Section 4.2, in some cases all the interactive quantities can not be selected as state variables. This case is addressed in this section.

4.4.1 Motivating example

The model of a perfect gas is shown in Figure 4.2. 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 parameters, i.e. time-independent properties of the physical model. The number of gas moles (n), the internal energy (U), the gas pressure (P) and the gas temperature (T) are time-dependent variables of the physical model.

The evolution in time of the *time-dependent quantities* is described by the physical model equations. As was discussed in Section 4.2, the *time-dependent quantities* have to be selected as state variables in order to be interactive quantities, i.e, their value can be changed abruptly as a result of the user's interaction.



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

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

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

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

$$C_P - C_V = R$$

Figure 4.2: Model of a perfect gas.

In general, different choices of the model state-variables are possible. As the model of a perfect gas in a fixed volume has two degrees of freedom, only two variables can be simultaneously selected as state variables. Possible choices in the model shown in Figure 4.2 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.

The state variable selection should be made so that it includes all the interactive quantities. If the user wants to interactively change 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 if he wants to change T and to keep constant p. Likewise, the appropriate choice is e_2 if the user wants: (1) to interactively modify 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, i.e. time-independent properties of the physical model, 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 4.2. 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

in 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.

4.4.2 Model description

An approach to implement this capability is the following. Building the interactive model as composed of several instances 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 (the one with the chosen state selection) for executing the instantaneous change in the parameters and state variables, and for solving the re-start problem.

Modelica capability for state-selection control allows easy implementation of this approach (Otter & Olsson 2002). Three instantiations of the perfect-gas model (i.e., *perfectGas*) have been defined (see Figure 4.3):

1. *perfectGasSS1*, with $e = \{p, T\}$.
2. *perfectGasSS2*, with $e = \{n, T\}$.
3. *perfectGasSS3*, with $e = \{n, p\}$.

The Modelica code of the perfect-gas model is listed in Appendix B.

View-model connection

The schematic description of the model-view connection is shown in Figure 4.3. There are seven input signals to the model and one output signal. The function of these signals is explained below. The perfect gas virtual-lab will be used to illustrate the discussion.

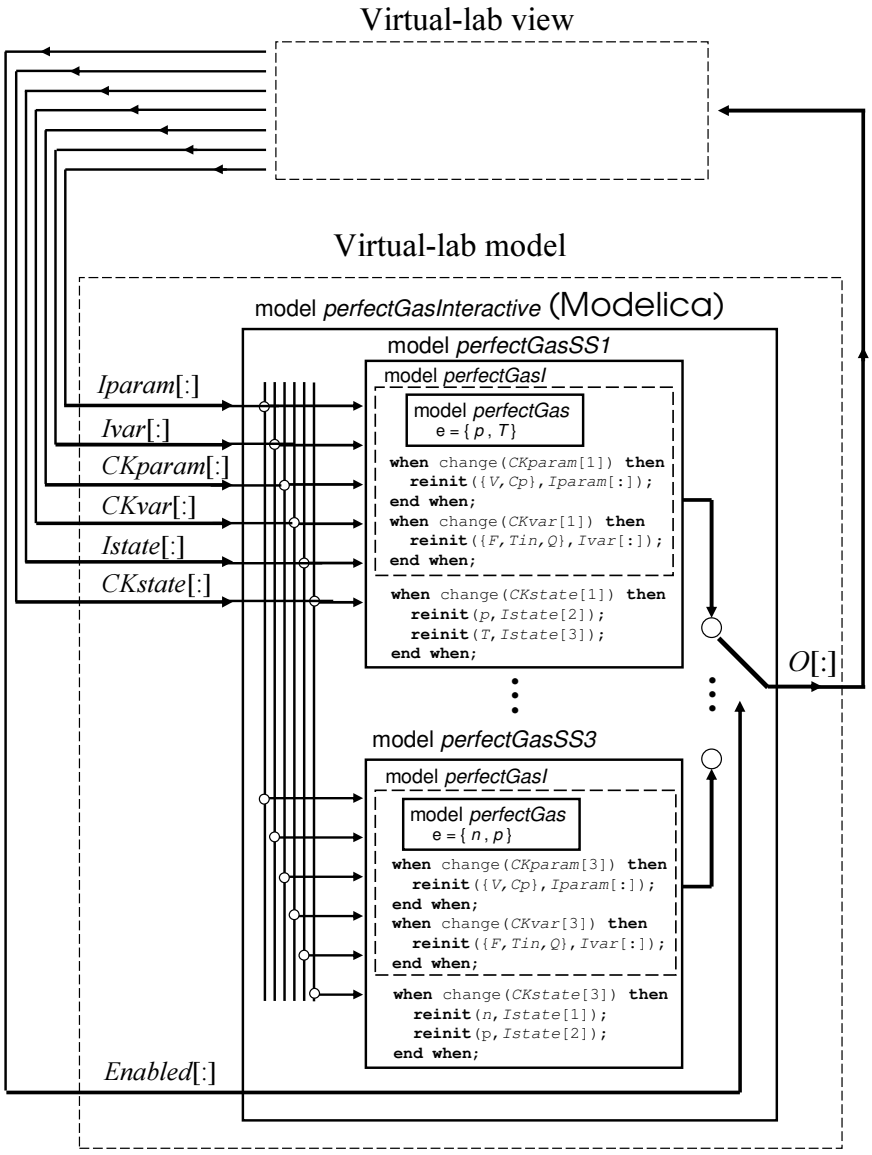


Figure 4.3: Schematic description of the model-view connection.

Interactive state variables

Two input variables to the model are used to carry out the interactive changes in the state: *Istate*[:] and *CKstate*[:] (see Figure 4.3).

- 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 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 4.3).

For instance, the virtual-lab view changes the value of the signal $CKstate$ [1] and updates the value of the vector $Istate$[:] when *perfectGasSS1* model is enabled ($Enable = [1, 0, 0]$) and a state variable value is changed by the user.

Interactive parameters and input variables

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 model. Their values are changed by using the *reinit* operator. Four input variables to the model are used (see Figure 4.3):

- Two arrays (*Iparam*[:], *Ivar*[:]) containing the new values.
- Two arrays (*CKparam*[:], *CKvar*[:]) for triggering the re-initialization events.

For instance, the virtual-lab view changes the value of the signal $CKparam$ [1] and updates the value of the vector $Iparam$[:] when *perfectGasSS1* model is enabled ($Enable = [1, 0, 0]$) and a parameter value is changed by the user.

Changing the state variable selection

When the user changes the state selection, the physical model instantiation corresponding to the new state choice must be re-initialized to start its trajectory at the last point described by the physical model instantiation corresponding to the previous state selection. To this end, the virtual-lab view:

1. Sets the new value of $Enable$[:].

2. Changes the value (from one to zero or vice-versa) of $CKparam\,[i]$, $CKstate\,[i]$ and $CKvar\,[i]$, where i is an integer whose value depends on the state selection (for instance, $i = 1$ if the new state selection is the one corresponding to *perfectGasSS1*).

3. Updates the value of the vector $Istate[:]$.

Output variables

The output variable array of the model, $O[:]$, contains the value of the variables linked to the properties of the virtual-lab *view*. The virtual-lab view uses the value of this output array ($O[:]$) to refresh the simulation view.

The value of the input array $Enabled[:]$ is set by the virtual-lab view. It selects which output is connected to the output signal $O[:]$. The output array in the perfect-gas model is the following: $O\,[:] = \{n, p, T, V, C_P, T_{in}, F, Q\}$.

4.5 Case study: tank system

The tank model shown again in Figure 4.1 is used to illustrate the previous discussion. Possible choices of the state variables include:

$$e_1 = \{h\} \qquad e_2 = \{V\} \qquad e_3 = \{F\}$$

where e_i represents one particular choice of the state variables.

This virtual-lab is required to support three ways of describing the interactive changes in the amount of liquid contained in the tank:

1. Changes in the liquid volume (V).

2. Changes in the liquid level (h).

3. Changes in the output flow of liquid (F).

In other words, each time the user needs to change the amount of liquid, he can choose among describing it in terms of the volume, in terms of the level, or in terms of the output flow. Different choices are possible during a given simulation

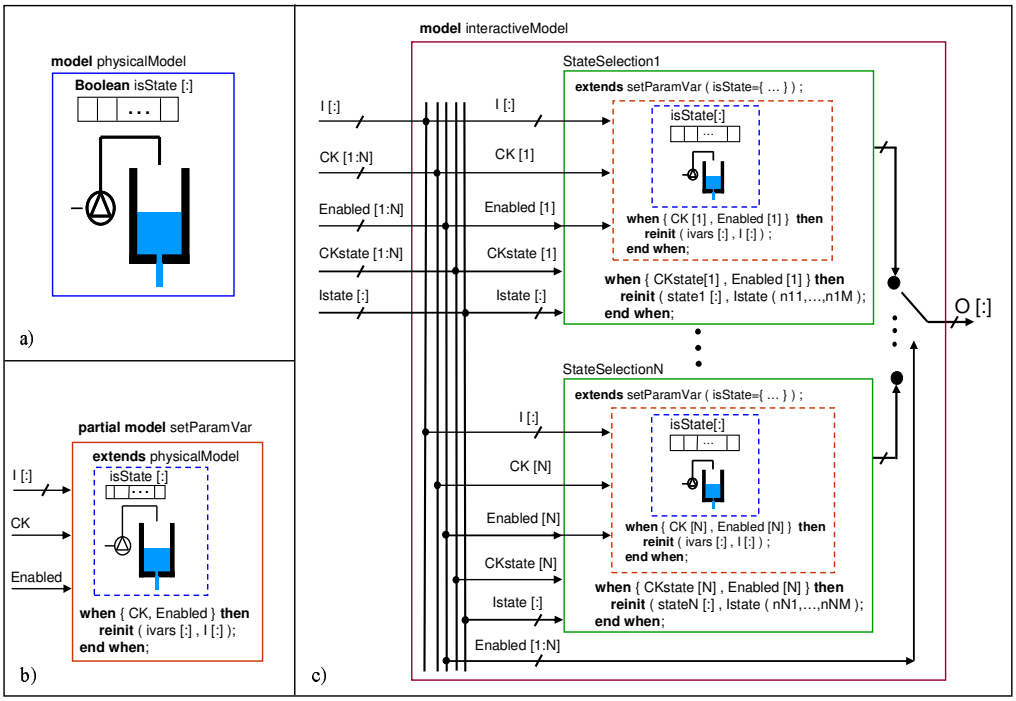


Figure 4.4: Schematic description of the proposed modeling methodology for interactive simulation.

run. However, V, h and F can not be simultaneously selected as state variables. The approach proposed in Section 4.4.2 is applied with slight modifications:

1. The interactive model is composed of as many instances of the physical model as different state selections are required. In this case, three selections of the state variables are required: $e_1 = \{h\}$, $e_2 = \{V\}$ and $e_3 = \{F\}$. The boolean vector isState[:], declared in physicalModel, allows controlling the state selection. The size of this vector is equal to the number of interactive time-dependent quantities. For instance, if isState[:] is set to the value {false,true,false} when instantiating the physical model, then the liquid volume (V) is selected as a state variable. Also, the interactive parameters (A, a) and the input variable (v) have been defined as constant state variables (see Modelica Code 4.4). This first step in the implementation of the interactive model is represented in Figure 4.4a.

2. The setParamVar class is defined (see Figure 4.4b). It inherits from physicalModel, and it contains the *when*-clauses required to change the value of the

```
model tank
  parameter Boolean hIsState = false;
  parameter Boolean VIsState = false;
  Real h (stateSelect = if hIsState
                        then StateSelect.always else StateSelect.never)
                        "Liquid level";
  Real V (stateSelect = if VIsState
                        then StateSelect.always else StateSelect.never)
                        "Liquid volume";
  parameter Real Ainitial "Initial value of the tank section";
  Real A (start = Ainitial) "Tank section - Interactive quantity";
  ...
equation
  der(A) = 0;
  ...
end tank;

model pipe
  Real F (stateSelect = if FIsState
                        then StateSelect.always else StateSelect.never)
                        "Liquid flow";
  parameter Real aInitial = 1 "Initial value of the pipe section";
  Real a (start = aInitial) "Pipe section - Interactive quantity";
  ...
equation
  der(a) = 0;
  ...
end pipe;

model pump
  parameter Real vInitial "Initial value of the applied voltage";
  Real v (start = vInitial) "Voltage applied to the pump - Interactive";
  parameter Real kInitial "Initial value of the pump parameter";
  Real k (start = kInitial) "Pump parameter - Interactive quantity";
  ...
equation
  der(v) = 0;
  der(k) = 0;
  ...
end pump;

partial model physicalModel
  parameter Boolean[3] isState;
  tank tank1 ( hIsState = isState[1],
               VIsState = isState[2], ...);
  pipe pipe1 ( FIsState = isState[3], ...);
  pump pump1 ( ...  );
  ...
end physicalModel;
```

Modelica Code 4.4: Tank model with three different selections of the state variables.

interactive parameters and input variables. These interactive quantities are represented by the ivars[:] array, and their new values, specified interactively by the virtual-lab user, are represented by the I[:] array (note that in this example: I = { Iparam, Ivar }). The size of these arrays is equal to the number of interactive parameters plus the number of interactive input variables. The *when*-clauses are triggered by the boolean variables CK and Enabled. When the value of any of these two variables changes from *false* to *true*, then the ivars[:] array is re-initialized to the value of the I[:] array.

3. There are defined as many components (StateSelection1, ..., StateSelectionN) as different state-variable choices are required (e_1 = state1[:], ..., e_N = stateN[:]). The number of state-variable choices to be supported by the virtual-lab is represented by N. The class of these components inherits from setParamVar (see Figure 4.4c). In addition, it contains the *when*-clauses required to re-initialize its state-variable array (i.e., state array) to the values interactively set by the user (i.e., Istate array).

   The CK[1:N] and Enabled[1:N] arrays trigger the re-initialization of the interactive parameters and input variables (note that in this example: CK = { CKparam, CKvar }). The CKstate[1:N] and Enabled[1:N] arrays trigger the re-initialization of the interactive time-dependent quantities. The $i-th$ component of these arrays controls the $i-th$ instantiation of the physical system (i.e., StateSelection$i$).

   The array Enabled[1:N] indicates which state-variable selection is enabled. It is used to select which output is connected to the output variables (O[:]). These are the variables used to refresh the virtual-lab view.

4.6 Conclusions

A novel modeling methodology, oriented to adapt any Modelica model for *runtime* interactive simulation, has been discussed and it has been applied for programming JARA 2i.

5

Virtual-labs Implemented by Combining Ejs, Matlab/Simulink and Modelica/Dymola

5.1 Introduction

The implementation of virtual-labs supporting *runtime* interactivity by the combined use of Ejs, Matlab/Simulink and Modelica/Dymola is proposed. The virtual-lab *model* is programmed using the Modelica language and translated using Dymola. The *view* is developed using Ejs. The *model-view communication* is implemented using the following interfaces:

- *Modelica/Dymola to Matlab/Simulink interface.* The C-code generated by Dymola for the Modelica model can be embedded within a Simulink block (Dynasim 2006).
- *Ejs to Matlab/Simulink interface.* On the other hand, Ejs allows the model to be partially or completely developed using Simulink block diagrams (Sanchez et al. 2005a,b).

This approach allows taking advantage of the best features of each tool:

- Ejs 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.

– Matlab/Simulink capabilities for modeling of automatic control systems and for model analysis.

5.2 Virtual-lab model

The methodology proposed in Chapter 4 has to be applied in order to adapt the physical model for *runtime* interactive simulation. In addition, the following two model modifications have to be carried out:

1. As the Modelica model has to be embedded within a Simulink block, the computational causality of the model-view interface variables has to be explicitly set.

2. User's interactive actions generate abrupt changes in the value of the interactive variables. The code (i.e., *when* clauses) to implement these interactive changes has to be included in the model.

5.3 Virtual-lab view

The virtual-lab view is implemented using Ejs. Ejs includes a panel for the view description, which is divided in two parts (see Figure 5.1):

– An area containing the Ejs' "view elements".

– An area named "Tree of elements". The view is composed by instantiating and connecting with the mouse the "view elements" in this area.

The tree of elements of the perfect-gas virtual-lab is shown in Figure 5.1.

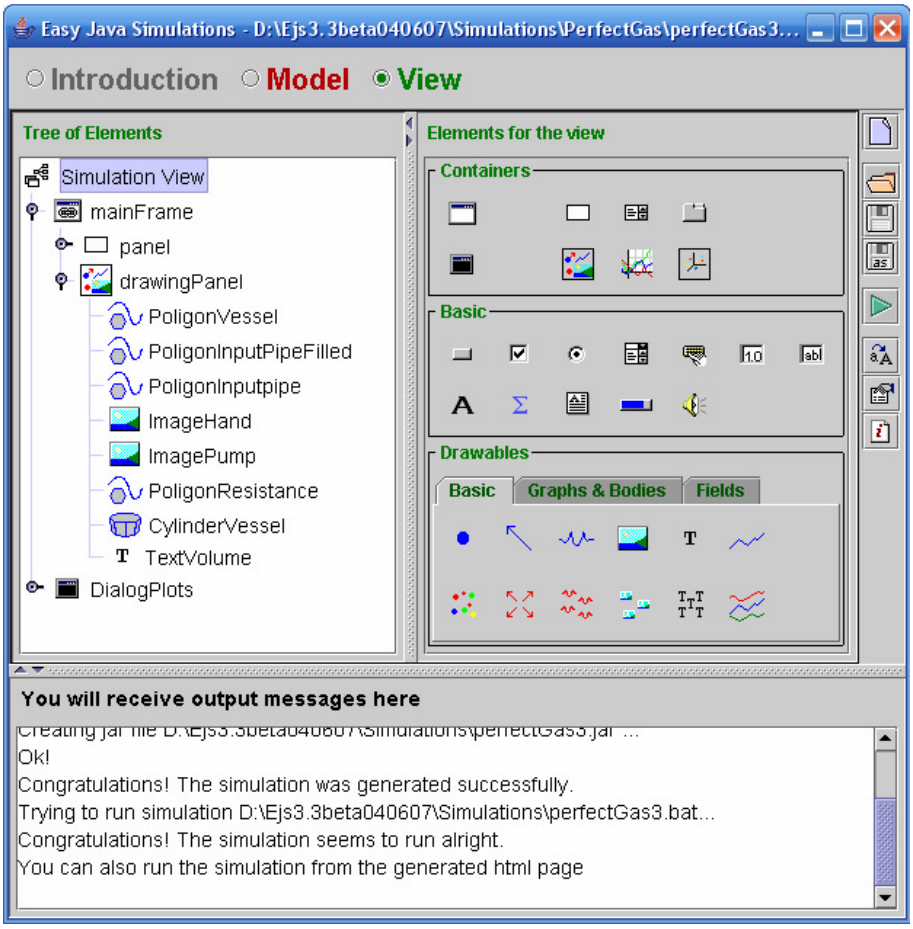


Figure 5.1: View description of the perfect-gas virtual-lab.

5.4 Virtual-lab set up

The perfect gas model described in Section 4.4.1 is used to illustrate the implementation of the model-view communication through Matlab/Simulink (Sanchez et al. 2005a,b). The Simulink model of the perfect-gas is shown in Figure 5.2a:

- 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.

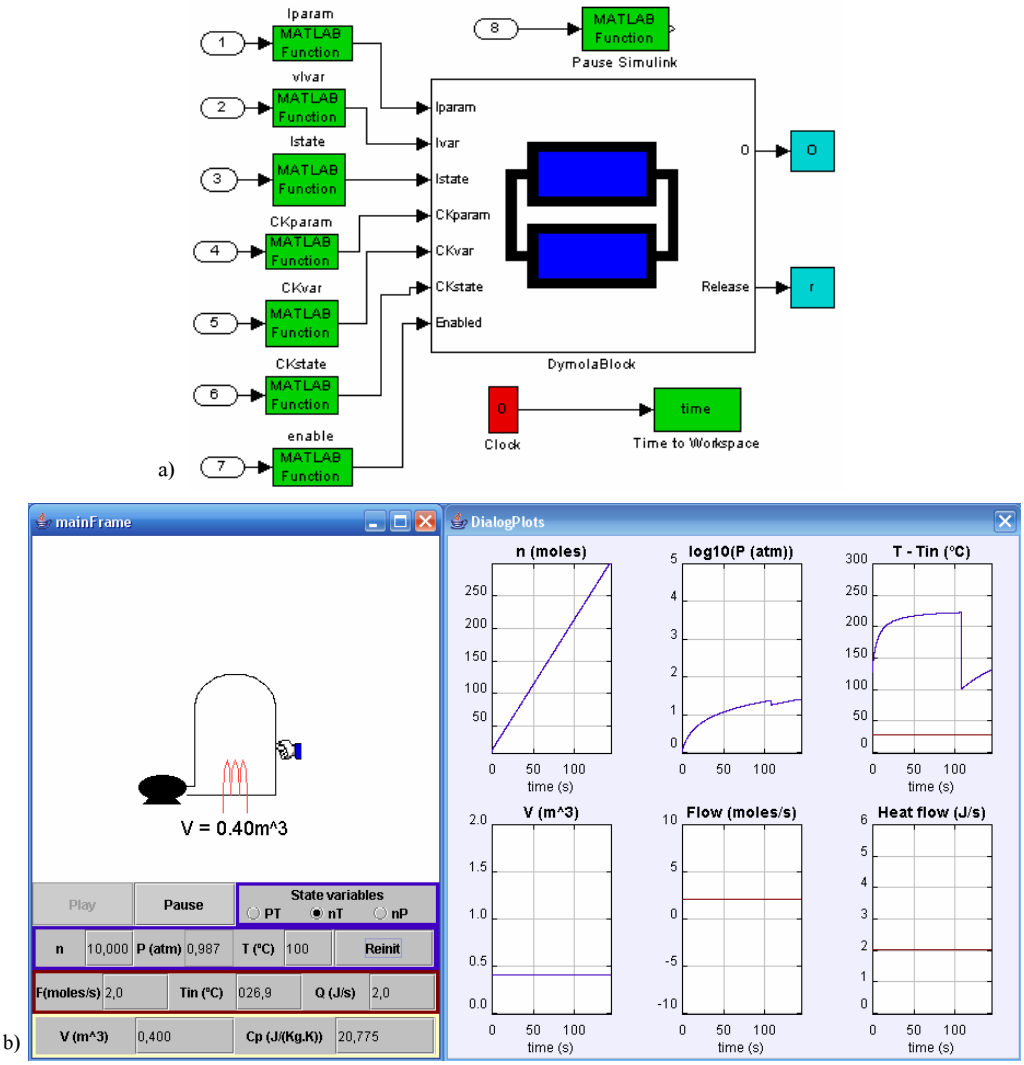


Figure 5.2: Perfect-gas virtual-lab: a) Simulink model; and b) view.

– 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. The virtual-lab view (programmed in 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 5.2b. 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 to 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.

The dynamic response of the perfect gas to a step change in the gas temperature is shown in Figure 5.2b. This change has been interactively performed by the virtual-lab user at the simulated time 108 s. The state selection is $e = \{n, T\}$. The following six plots are shown in Figure 5.2b: (1) the number of moles; (2) the decimal logarithm of the gas pressure; (3) the value of the gas temperature and the gas flow temperature; (4) the volume of the recipient containing the gas; (5) the liquid flow rate generated by the pump; and (6) the heat flow rate.

5.5 Case study I: quadruple-tank process virtual-lab

The quadruple-tank process is represented in Figure 5.3 (Johansson 2000). It can be used to teach different aspects of the multivariable control theory (Johansson 2000, Dormido & Esquembre 2003). The goal is to control the level of the two lower tanks (h_1 and h_2) by manipulating the pump voltage (v_1 and v_2).

5.5.1 Virtual-lab model

In order to illustrate their different dynamic behavior, two different models of the process have been implemented: a linear model and a non-linear model. The non-linear model has been composed by using the *tankProcessLAB* Modelica library (see Figure 5.4a). Mass balance and Bernoulli's law are applied to model the tanks and the flows. The Modelica diagram of the physical model is shown in Figure 5.4b.

The implementation of the *tankProcessLAB* Modelica library is part of the work developed in this dissertation. This library is composed of some basic models of hydraulic components (i.e., tanks, pipes, valves, etc.) that have been adapted

for interactive simulation. The *tankProcessLAB* library can be downloaded from http://www.euclides.dia.uned.es

The virtual-lab supports interactive changes in the tank physical parameters (i.e., cross-section, shape and cross-section of the outlet hole) and in the amount of liquid stored inside the tanks. Two selections of the state variables are supported: $e_1 = \{volume\}$ and $e_2 = \{level\}$. As a consequence:

- The changes in the stored amount of liquid can be defined in terms of the liquid level or the liquid volume.
- The tank cross-section and shape changes can take place under one of the following alternative conditions: (1) the liquid volume inside the tank is kept constant; or (2) the liquid level is kept constant.

Two different control strategies have been implemented: manual control and decentralized PID. The switching between these two control strategies can take place during the simulation run. The parameters of the PID controllers can be changed interactively.

5.5.2 Virtual-lab set up

The Simulink model containing the *DymolaBlock* block is shown in Figure 5.5. Observe that the structure of this Simulink model is analogous to the perfect-gas model, shown in Figure 5.2a.

The virtual lab is shown in Figure 5.6. The main window (on the left side of Figure 5.6) 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 liquid levels, the tank cross-sections and the level setpoints represented in the schematic diagram are linked to the respective model variables: their values can be interactively changed by dragging with the mouse.

The sliders placed under the schematic diagram allow interactively changing the pump voltages (v_1 and v_2) and the valve settings (g_1 and g_2). The radio-buttons allow choosing the state variables (liquid volumes or levels) and the

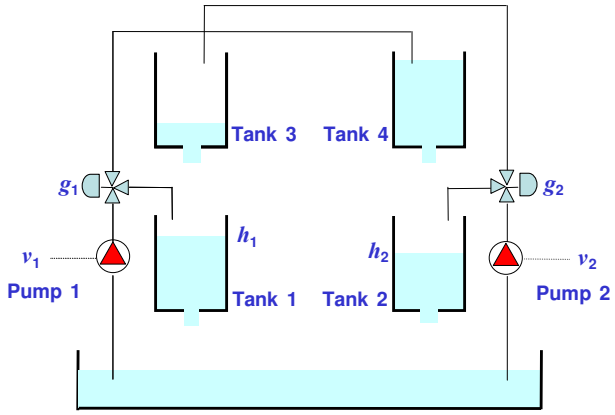
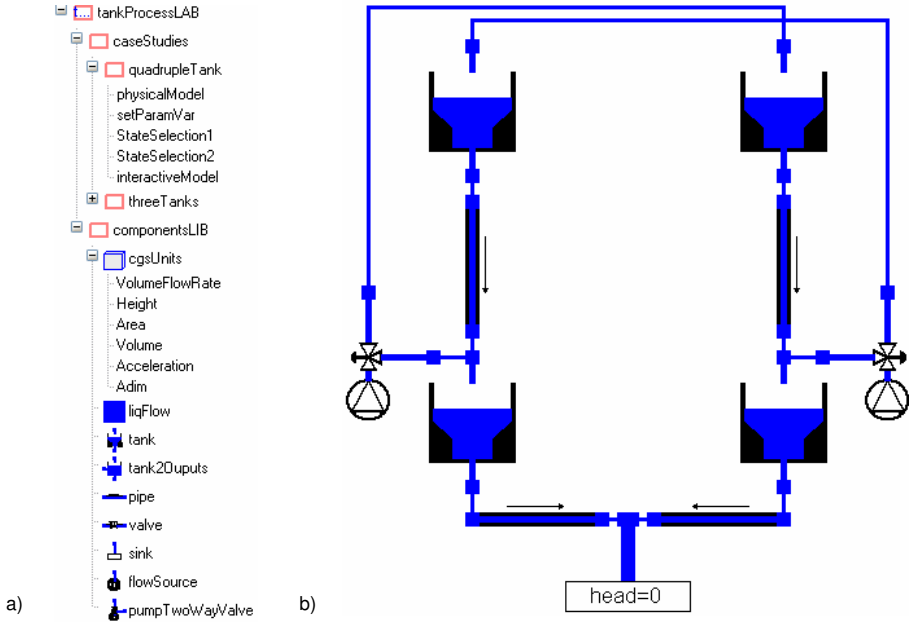


Figure 5.3: Schematic representation of the quadruple-tank process.



a) b)

Figure 5.4: Quadruple-tank process: a) *tankProcessLAB* Modelica library; and b) diagram of the quadruple-tank Modelica model.

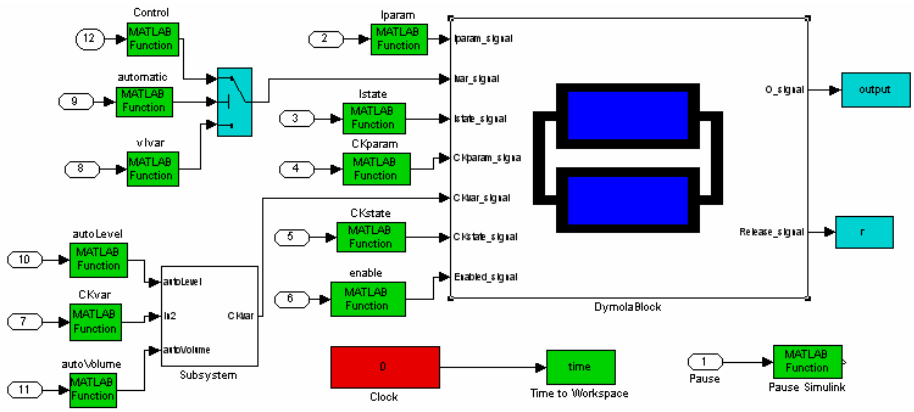


Figure 5.5: Simulink model of the quadruple-tank process virtual-lab.

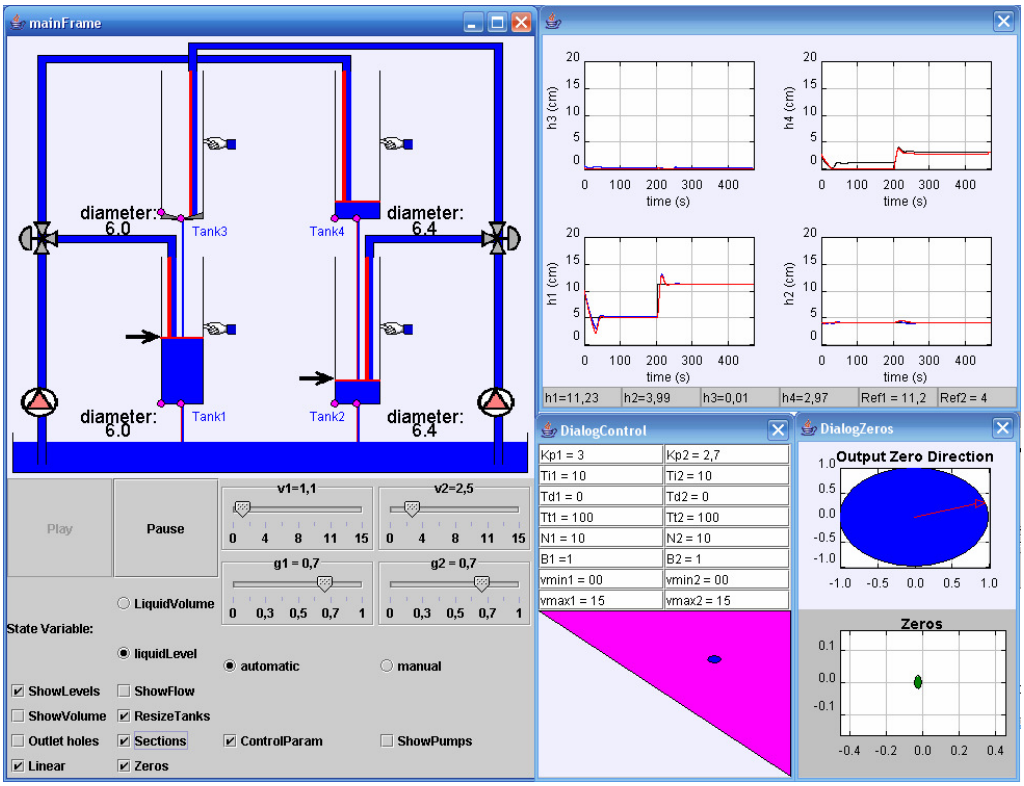


Figure 5.6: View of the quadruple-tank process virtual-lab.

control strategy (manual or decentralized PID). The "Linear" box shows and hides the liquid levels calculated from the linear model simulation.

Clicking the "ResizeTanks" and "Sections" boxes bring-in two graphical gadgets (in the form of a hand and of control circles, respectively) that can be dragged to change the diameter and shape of the tank section. The "Outlet holes" box opens and closes a secondary window, where the user can interactively modify the diameter of the outlet holes.

The "DialogZeros" and "DialogControl" windows are displayed by clicking on the "Zeros" and "ControlParam" boxes respectively (see Figure 5.6). The "DialogZeros" window shows the zero location and its directionality (Skogestad & Postlethwaite 1996). The "DialogControl" window allows changing interactively the PID parameters and the position of the point (g_1, g_2). This position has important consequences: above the diagonal (i.e., $g1 + g2 > 1$) the system is minimum phase (easy control problem), and below it is non-minimum phase (difficult control problem).

The rest of the check-boxes open and close graphic plots of the liquid levels, volumes and flows, and plots of the voltage applied to the pumps. Some of these plots are displayed on the right side of Figure 5.6. The dynamic response of the four tank system to a step change in the setpoint of the tank 1 liquid level from 5 cm to 11.2 cm is shown in Figure 5.6. This change has been interactively performed by the virtual-lab user at the simulated time 204.6 s. The system is operating in automatic mode.

5.6 Case study II: chemical reactor virtual-lab

The physical model of the chemical reactor has been composed using the JARA 2i Modelica library. The interactive model has been implemented by extending the physical model described in Section 2.5.3 and by including the required code to: (1) be useful as a Simulink block; and (2) implement the user's changes in the value of the interactive quantities. The Modelica code of the interactive model is included in Appendix B.

The Modelica models of the chemical reactor and the controllers are embedded within the *SystemBlock* and *PIDBlock* blocks respectively (see Figure 5.7). The virtual-lab view is shown in Figure 5.8.

The main window (on the left side of Figure 5.8) 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 user can interactively choose between manual and automatic control. The automatic control is intended to perform the following operation policy (see Figure 5.9):

1. Fill up the reactor with the reacting liquid. The inflow is controlled by a PID.

2. Preheat to certain temperature, and let the reaction proceed adiabatically.

3. Start cooling when either the maximum allowable reaction temperature occurs or the desired conversion is reached, and cool down to the desired temperature.

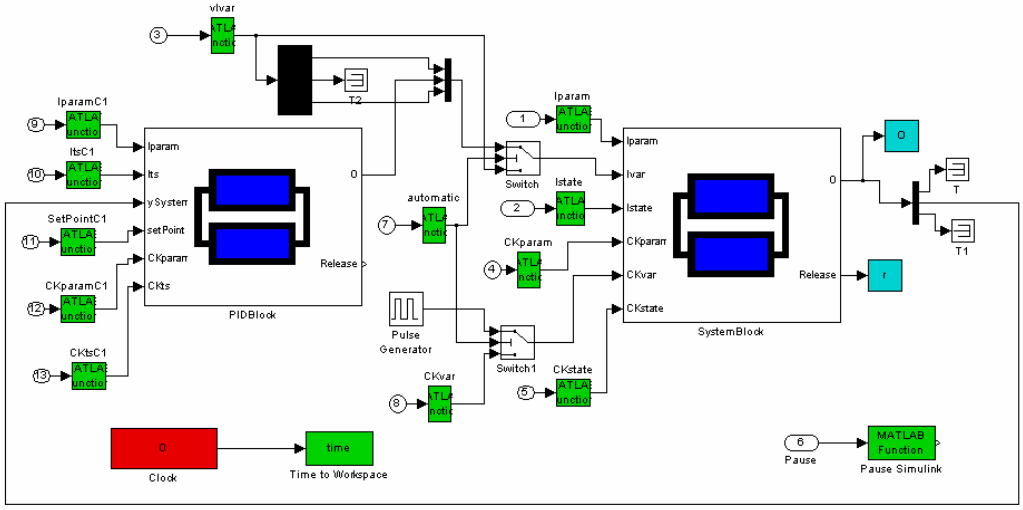


Figure 5.7: Simulink model of the chemical reactor virtual-lab.

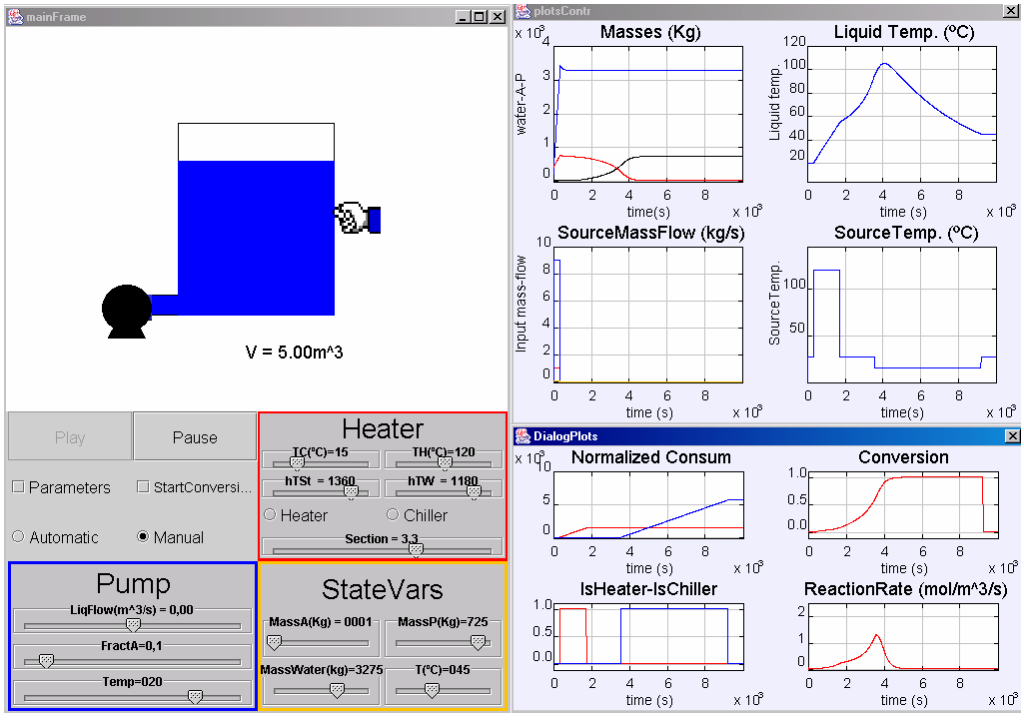


Figure 5.8: View of the chemical reactor virtual-lab.

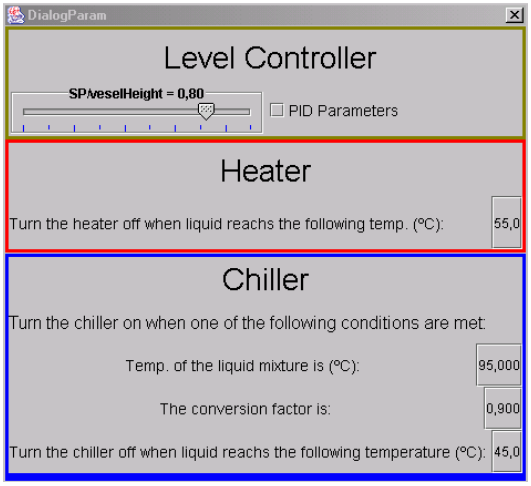


Figure 5.9: Window menu to determine the operation policy of the chemical reactor virtual-lab.

4. Empty the reactor.

The value of the PID-controller parameters, the temperatures defining the operation policy and the desired conversion can be changed interactively. Also, the value of the model state-variables (i.e., the temperature and mass of the reaction mixture, and the concentration of A and P), the model parameters (i.e., the reactor volume and section, the area of the heat exchanger, and the physical-chemical data of the steam and cooling water), and the input variables (i.e., the inflow temperature and concentration) can be changed interactively during the simulation run. The secondary windows on the right side of Figure 5.8 contains plots showing the evolution of some relevant process variables.

5.7 Case study III: industrial boiler virtual-lab

A boiler virtual-lab has been implemented by the combined use of Ejs, Simulink and Modelica/Dymola. The physical model of the industrial boiler has been composed using the JARA 2i Modelica library (see Section 2.5.4). The interactive model has been implemented analogously to the chemical reactor model (see Section 5.6). That is, it has been implemented by extending the physical model described in Section 2.5.4 and by including the required code to: (1) be useful

as a Simulink block; and (2) implement the user's changes in the value of the interactive quantities.

The Simulink model and the Ejs view of the boiler virtual-lab are shown respectively in Figures 5.10 and 5.11. The user can interactively choose between two control strategies: manual and decentralized PID. The control system has been modeled using Modelica: a PID is used to control the water level and another PID is used to control the vapor flow. The manipulated variables are the pump water-flow and the heater heat-flow respectively. The parameters of these PID controllers can be changed interactively. In addition, the value of the model state-variables (mass and temperature of the water and the vapor), parameters (inner volume of boiler), and input variables (temperature of the input water, valve opening and output pressure) can be changed interactively during the simulation run.

The dynamic response of the industrial boiler to a step change in the setpoint of the vapor output flow from 8 to 9.2 moles/s is shown in the right window of Figure 5.11. This change has been interactively performed by the virtual-lab user at the simulated time 201.8 s. The boiler is operating in automatic mode.

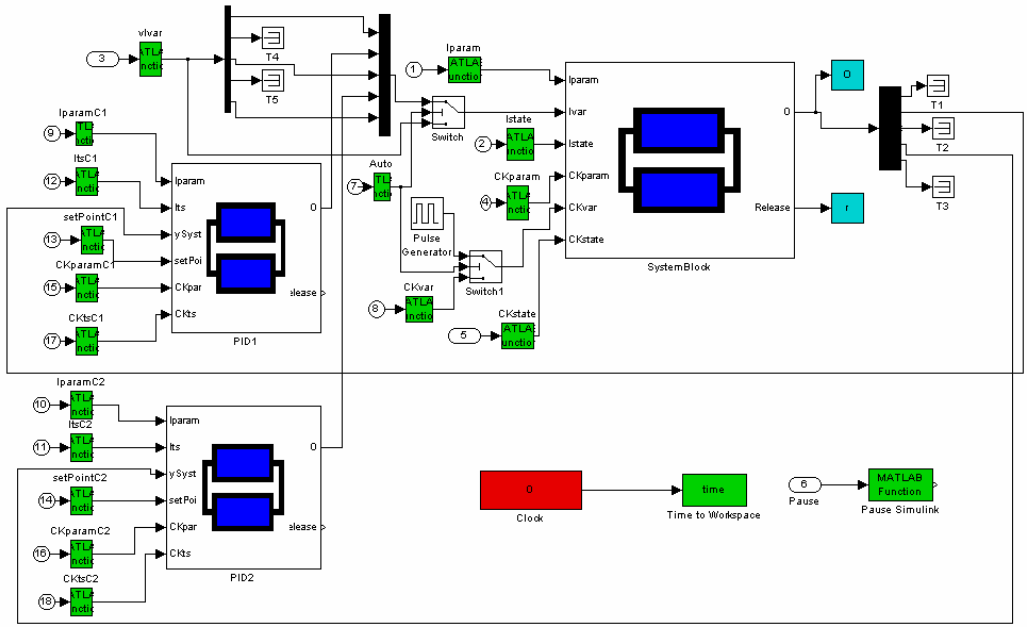


Figure 5.10: Simulink model of the industrial boiler virtual-lab.

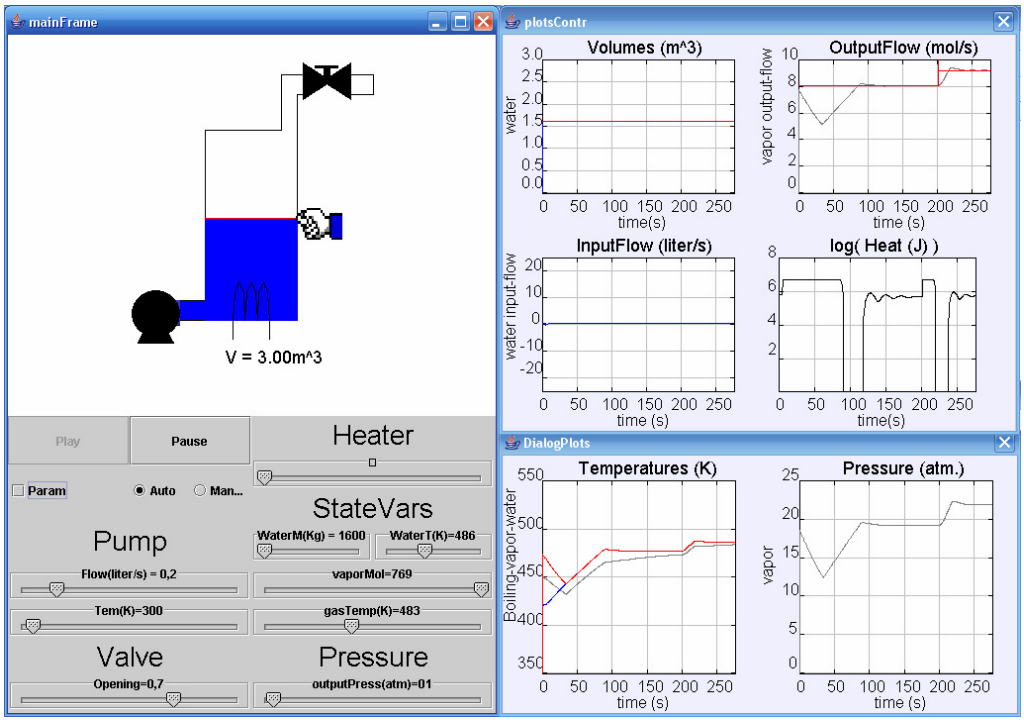


Figure 5.11: View of the industrial boiler virtual-lab.

5.8 Case study IV: heat-exchanger virtual-lab

The physical model of the heat-exchanger has been composed using the JARA 2i Modelica library (see Section 2.5.5). The interactive model has been implemented analogously to the chemical reactor (see Section 5.6) and industrial boiler (Section 5.7) models. That is, it has been implemented by extending the physical model described in Section 2.5.5 and by including the required code to: (1) be useful as a simulink block; and (2) implement the user's changes in the value of the interactive quantities.

The Simulink model is shown in Figure 5.12. The interactive model of the heat exchanger, written in Modelica language, has been embedded within the *DymolaBlock* block.

The view of the virtual-lab is shown in Figure 5.13. 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.

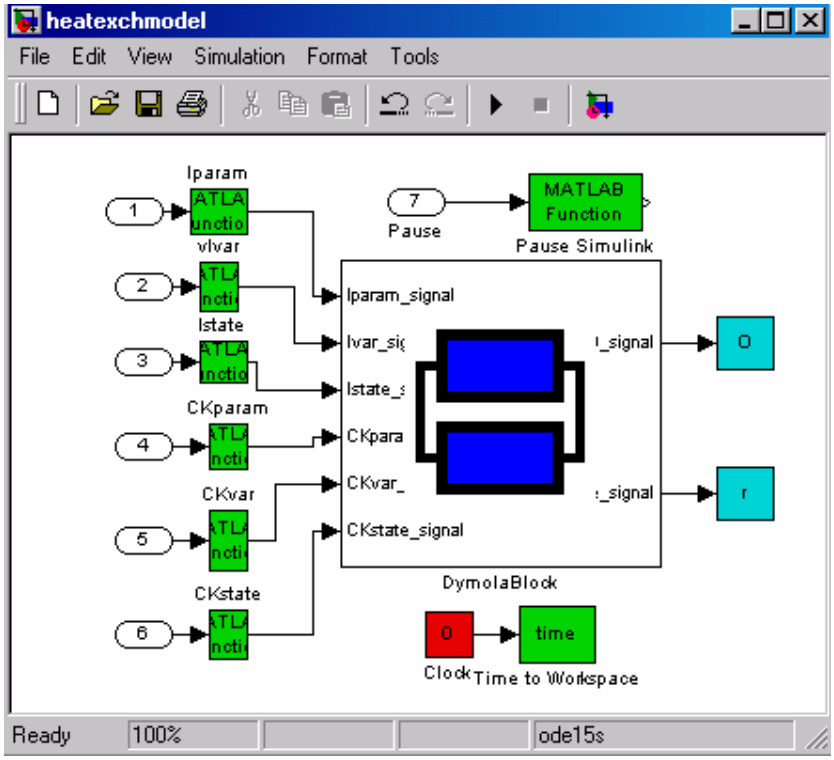


Figure 5.12: Simulink model of the heat exchanger virtual-lab.

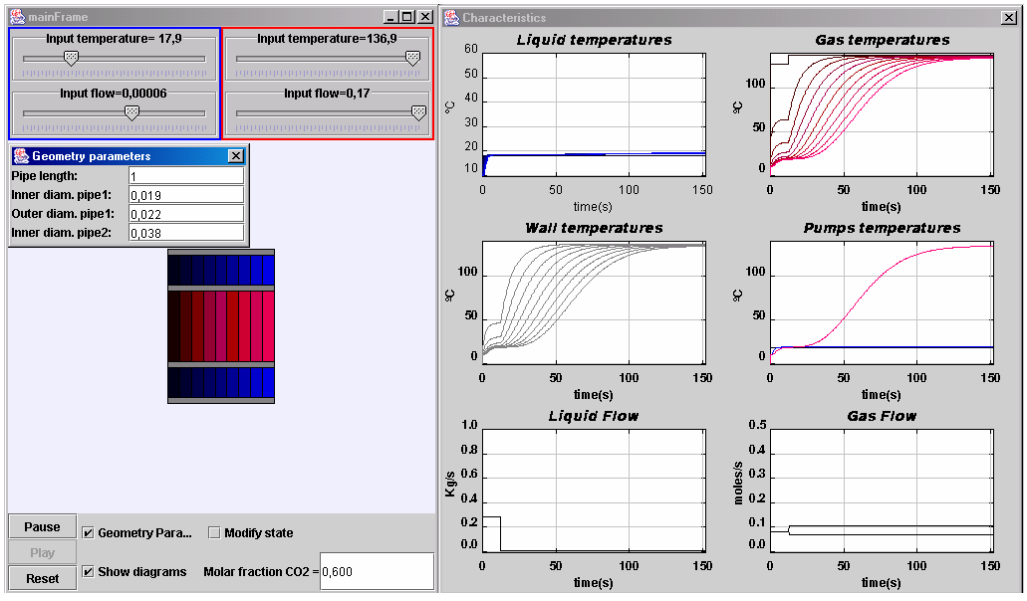


Figure 5.13: View of the heat exchanger virtual-lab.

5.9 Conclusions

The feasibility of combining Modelica/Dymola, Matlab/Simulink and Ejs for implementing *runtime* interactive simulations has been demonstrated. The use of Modelica language has reduced considerably the modeling effort and it has permitted better reuse of the models. Ejs' visual elements have allowed easy creation of the virtual-lab view. This approach has been successfully applied to setting up four virtual-labs intended for control education: the quadruple-tank process, the chemical reactor, the industrial boiler, and the heat-exchanger virtual-lab.

6

VirtualLabBuilder Modelica Library - User's Perspective

6.1 Introduction

A fundamental goal of this research work is to facilitate the description and implementation of virtual-labs using only the Modelica language. To achieve this goal, a Modelica library has been designed and programmed. This library, named *VirtualLabBuilder*, contains Modelica models implementing graphic interactive elements, such as containers, animated geometric shapes, basic elements and interactive controls. These models allow the virtual-lab developer:

1. To compose the virtual-lab view.
2. To link the visual properties of the virtual-lab view with variables of the virtual-lab model.
3. To link HTML pages to the virtual-lab view. These HTML pages are intended to serve as virtual-lab user's documentation.

The discussion about *VirtualLabBuilder* design and use has been structured into Chapters 6 and 7:

– A library description oriented to the virtual-lab developers and some cases of use are provided in Chapter 6.

– Details about the design and implementation of the library, that might be of interest to the *VirtualLabBuilder* developers, are discussed in Chapter 7.

Finally, the feasibility of setting up virtual-labs of complex Modelica models by using *VirtualLabBuilder* is demonstrated in Chapter 8. For that purpose, a virtual-lab showing the thermodynamic behavior of an experimental house has been implemented. This model was developed by M. Weiner as part of his M.S. thesis (Weiner 1992, Weiner & Cellier 1993).

6.2 Design objectives

The purpose of the *VirtualLabBuilder* library is to facilitate the implementation and execution of a virtual-lab completely described in Modelica language. The following objectives have been taken into account for the design of the library:

1. To have a set of Modelica classes representing each one a graphic component displayed by the virtual-lab view.
2. To allow easy description of the virtual-lab view, using an object oriented methodology, and to be able to describe complex virtual-lab views.
3. To automatically generate the executable code of the virtual-lab view.
4. To automatically generate, in a way completely transparent to the user, the code required to perform the runtime communication between the virtual-lab model and view.

6.3 Overview of the proposed approach

The virtual-lab definition includes the description of the introduction, the model, the view, and the bidirectional flow of information between the model and the view. The virtual-lab definition process is outlined next.

1. **Virtual-lab model.** Any Modelica model can be transformed into other Modelica model suitable for interactive simulation. A systematic methodology to perform this transformation was proposed in Section 4.2. Essentially, it consists in modifying the model so that all the variables that need to be changed interactively during the simulation (i.e., the *interactive variables*) are formulated as state variables. In particular, parameters are redefined as time-dependent variables whose time-derivative is equal to zero. Input variables are reformulated analogously in order to become interactive variables. Modelica's *when* clause and *reinit* operator allow describing instantaneous changes in the value of the state variables. This feature is exploited in order to perform the instantaneous changes in the value of the interactive variables produced by the user's interaction. Some of these model manipulations could be performed automatically by a software tool. However, at the present time, they have to be carried out manually by the virtual-lab developer.

2. **Virtual-lab view.** The virtual-lab developer has to define a Modelica class describing the virtual-lab view. This class has to extend another class, named PartialView, that is included in *VirtualLabBuilder* library (see Figure 6.1a). The communication interval (i.e., time interval between to consecutive model-view communications) is a parameter of the PartialView class (T_{com}), that can be set by the virtual-lab developer. PartialView class contains a pre-defined component: the *root element* for the view description. The classes describing the graphic components are within the Containers, Drawables, InteractiveControls and BasicElements packages of *VirtualLabBuilder* library (see Figures 6.1b, 6.1c, 6.1e and 6.1f respectively). The virtual-lab designer has to compose the virtual-lab view class by instantiating and connecting the required graphic components. The graphic components have to be connected forming a structure, whose root is the *root element*. The connections among the graphic components determines their layout in the virtual-lab view. *VirtualLabBuilder*'s graphic components and their connection rules are discussed in Section 6.4.

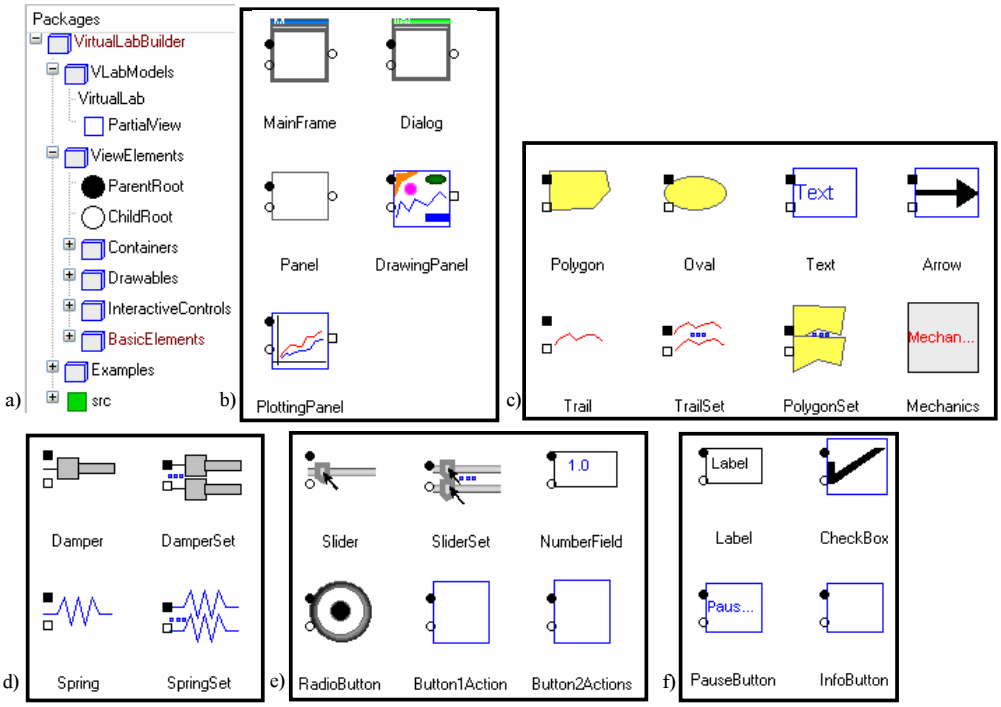


Figure 6.1: *VirtualLabBuilder* library: a) general structure; and classes within the following packages: b) Containers; c) Drawables; d) Mechanics; e) InteractiveControls; and f) BasicElements.

3. **Virtual-lab set up.** The virtual-lab developer has to define a Modelica class describing the complete virtual-lab. This class has to contain an instance of the VirtualLab class, which is within the *VirtualLabBuilder* library (see Figure 6.1a). VirtualLab class has the following parameters: the model-to-view communication interval (T_{com}), the name of the Java file (the content of this file is generated during the model initialization process), the class describing the virtual-lab model, and the class describing the virtual-lab view (see Figure 6.2). These two classes have been programmed in Steps 1 and 2 respectively. The virtual-lab designer has to set the value of these parameters by writing the name of these two classes. In addition, he has to specify how the variables of the model and the view Modelica classes are linked. This is accomplished by writing the required Modelica equations inside the Modelica class defining the complete virtual-lab.

4. **Virtual-lab translation and execution.** The virtual-lab developer needs to translate using Dymola (Dynasim 2006) an instance of the Modelica class

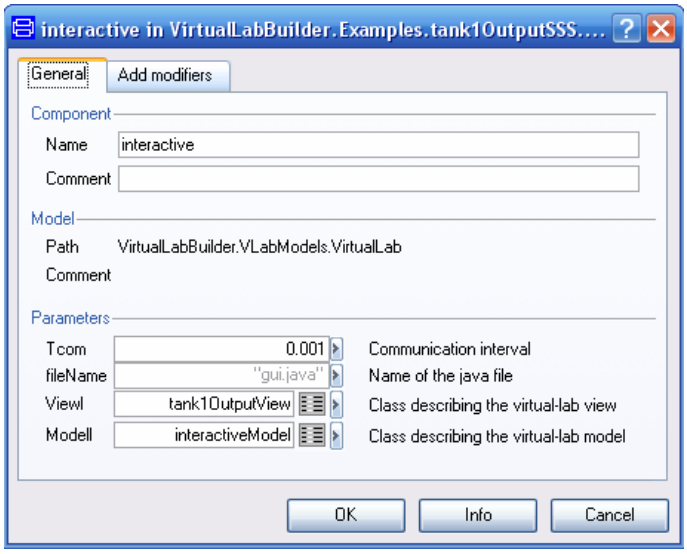


Figure 6.2: Parameter window of the *VirtualLab* class.

defined in Step 3 into an executable file (i.e., *dymosim.exe* file). The virtual-lab is started by executing this file.

5. **Automatic code generation and run.** At the beginning of the simulation run, some calculations are performed in order to solve the model at the initial time. The *initial sections* of the Modelica model describing the virtual-lab are evaluated. In particular, the *initial sections* of the interactive graphic objects composing the virtual-lab view class 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, packed into a jar file and executed. Also, the communication procedure between the model and the view is automatically set up. This communication is based on a client-server architecture: the C-program generated by Dymola (Dynasim 2006) (i.e., *dymosim.exe*, see Step 4) is the server and the Java program (which has been automatically generated during the model initialization) is the client. Once the jar file is executed, 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 model sends the data required to refresh the view and the view sends the value of the variables modified due to a user action at the time instant when the communication is performed. The time interval between two consecutive model-view communications was defined in Step 2.

6.4 *VirtualLabBuilder* library architecture

VirtualLabBuilder library is composed of the packages shown in Figure 6.1a. Some of them are intended to be used by the virtual-lab developers (i.e., *VirtualLabBuilder* users). These are:

1. ViewElements and VLabModels packages, which contain the classes required to implement the virtual-lab view and to set up the complete virtual-lab.

2. Examples package, which contains some tutorial material illustrating the library use.

The documentation of these packages is oriented to the *VirtualLabBuilder* users.

On the other hand, the classes within the src package are not intended to be directly used by the virtual-lab developers. The documentation of this package describes the implementation details required to modify and extend the *VirtualLabBuilder* library. In fact, the classes within ViewElements and VLabModels packages inherit from classes defined within src package, inheriting the structure and the behavior, and adding only the documentation oriented to the virtual-lab developer. The content of this package will be described in Section 7.2.

6.5 PartialView and VirtualLab classes

VLabModels package contains two classes: PartialView and VirtualLab. The purpose of PartialView and VirtualLab classes was briefly described in Section 6.3. PartialView

class has to be the super-class of the model defining the virtual-lab view. The class describing the complete virtual-lab has to contain an instance of VirtualLab class. Implementation details can be found in Chapter 7. ViewElements package is discussed in the next section.

6.6 Interactive graphic elements

ViewElements package contains the graphic elements that can be used to define the view. The *initial sections* of these elements contain calls to Modelica functions that perform calls to external C-functions. These C-functions write the Java code of the elements to a file, generating automatically the Java application (i.e., a *.jar* file) that is the virtual-lab view. The four packages included within ViewElements are described below.

6.6.1 Containers package

Containers package has those graphic elements that are intended to host other graphic elements. The container properties are set in the view definition and they can not be modified during the simulation run. *VirtualLabBuilder* contains the following five classes of containers (see Figure 6.1b):

- 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.
- Dialog class creates a window where containers and interactive controls can be placed. This class has only two differences with MainFrame class: (1) simulation run does not stop by closing this window; and (2) there can be more than one Dialog object.
- Panel class creates a panel where containers, interactive controls and basic elements can be placed.

– DrawingPanel class creates a two-dimensional container that only can contain drawable 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 values can be set by the user.

– PlottingPanel class creates a two-dimensional container with coordinate axes that only can contain drawable objects.

The MainFrame, Dialog and Panel classes have a parameter that specifies their layout policy. It sets where the elements placed within the element are located. Possible values are BorderLayout, GridLayout, HorizontalBox, VerticalBox and FlowLayout. Elements hosted inside a container that don't contain drawable objects have to specify their position (i.e., north, south, east or west) only if the layout policy of their container is BorderLayout.

6.6.2 Drawables package

Drawables package contains several classes implementing interactive 2-D shapes, whose properties (i.e., size, position, rotation angle, aspect ratio, color, etc.) can be linked to the model variables. They are intended to be used for building animated and interactive schematic representations of the system. These classes are: Polygon, PolygonSet, Oval, Text, Arrow, Trail and TrailSet (see Figure 6.1c). These elements draw a polygon, a set of polygons, an oval, a text, a vector, a trace, and a set of traces respectively.

Objects of Drawables classes must be placed inside containers that provide a coordinate system (i.e., containers of DrawingPanel and PlottingPanel classes).

In addition to this general-purpose interactive components, other domain-specific components can be implemented. In order to demonstrate this capability, the Mechanics package has been included within Drawables package (see Figure 6.1d). It contains four classes (i.e., Damper, DamperSet, Spring and SpringSet) implementing an interactive damper, a set of interactive dampers, an interactive spring and a set of interactive springs.

6.6.3 InteractiveControls package

InteractiveControls package contains classes that allow modifying interactively the value of the model variables. Each class includes the definition of an input real variable (*var*) and a boolean variable (*event*).

- The *event* variable is *true* at those time instants at which the interactive control is manipulated by the virtual-lab user. Otherwise, the *event* variable is *false*.
- The interactive model variable can be linked to the *var* variable by writing the corresponding equation.

This package contains the following classes:

- Slider class creates a slider.
- NumberField class creates an element that allows displaying and editing a numeric value.
- RadioButton class creates a radio-button.
- Button1Action class creates a button. The *var* variable is equal to one when the button is pressed and it is equal to zero otherwise. This variable can be used as a condition in a *when* clause. This way, the *when* clause is executed whenever the virtual-lab user presses the button.
- Button2Actions class creates a button. The *var* variable changes alternatively from zero to one and from one to zero whenever the button is pressed. By programming the corresponding *when* clauses, it is possible to associate two different actions to this button: an action is triggered when *var* changes from zero to one, and the other action is triggered when *var* changes from one to zero.
- SliderSet class creates a set of N sliders, where N is a class parameter. This class contains N instances of the Slider class. Each slider is connected to the next one following the connection rules described in Section 6.7.

6.6.4 BasicElements package

BasicElements package contains classes that can be hosted inside a window or a panel. This package contains the following classes:

- Label class creates a decorative label.
- CheckBox class creates a checkbox. The checkbox allows to show or hide the virtual-lab windows.
- PauseButton class creates a button that allows the user to pause or resume the simulation by clicking on it.
- InfoButton class creates a button that allows the user to show or hide a window displaying HTML pages. This feature allows including documentation in the virtual-lab. That is to say, it supports the implementation of the *virtual-lab introduction*.

6.7 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. The following two types of interfaces have been defined (see Figures 6.1b, 6.1c, 6.1d, 6.1e & 6.1f):

1. *Interface of container components.* It has three connectors (see Figure 6.1b). Two placed on one side (called "left connectors") and the third one (called "right connector") placed on the opposite side.
2. *Interface of interactive controls, basic elements and drawable elements.* It has two connectors (called "left connectors"): one filled and one empty (see Figures 6.1c, 6.1d, 6.1e & 6.1f).

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:

– If 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.

– If 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.

Example. The following example tries to illustrate how the graphic elements can be used to compose the view of a virtual-lab. In particular, the view of the tank process described in Section 4.2. The Modelica description of the virtual-lab view and the obtained virtual-lab are shown in Figure 6.3a and Figure 6.3b respectively. In this case, the model of the tank process has only one state selection and one state variable (the liquid level).

The mainFrame and dialog components are hosted inside root. The dPanel, panelS and panelN components are hosted inside mainFrame. The C component is hosted inside panelN. The pipe, vase, liqPipe and liquid components are hosted inside dPanel. The a, A, v and h components are hosted inside panelS. The plot component is hosted inside dialog. Finally, the component trail of the Trail class is hosted inside plot.

The window showing the component parameters is displayed by double clicking on the component icon. The parameter windows of the components trail, a and mainFrame are shown in Figures 6.4a, 6.4b and 6.4c respectively.

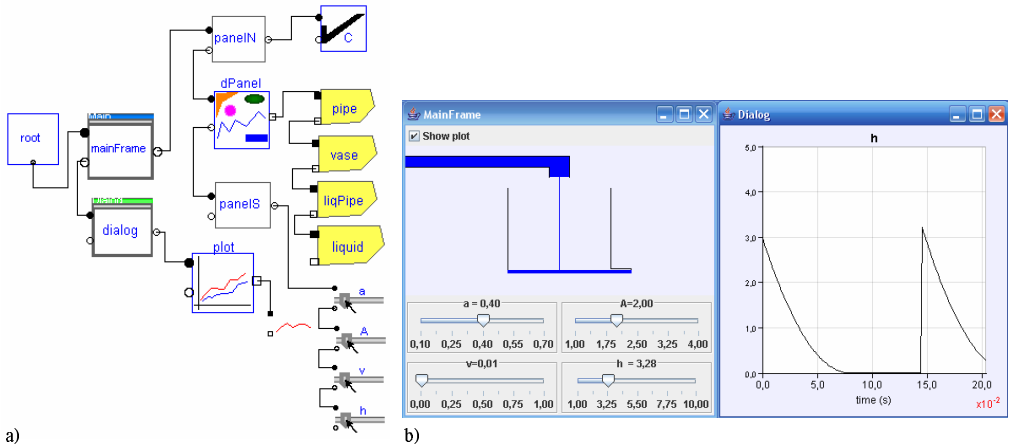


Figure 6.3: Tank process: a) Modelica description of the virtual-lab view; and b) virtual-lab.

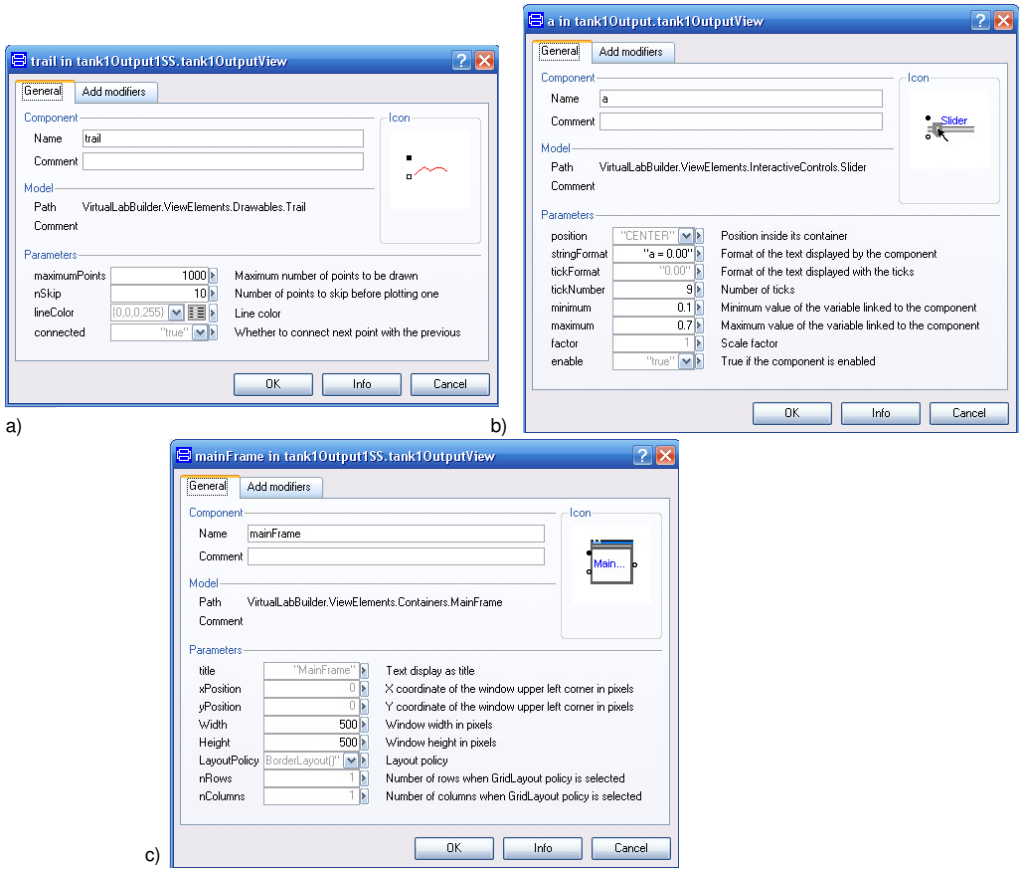


Figure 6.4: Parameter window of the following components: a) trail; b) a; and c) mainFrame.

6.8 Case study I: virtual-lab of an industrial boiler

The approach discussed in the previous sections is applied to the implementation of a virtual-lab for control education. This virtual-lab has been designed to illustrate the dynamic behavior of an industrial boiler operating under two different control strategies: manual and decentralized PID.

6.8.1 Virtual-lab model

The physical model of the industrial boiler has been composed using the JARA 2i Modelica library (see Section 2.5.4). The Modelica diagram of the boiler model is shown in Figure 6.5. The control system of the boiler is composed of two decoupled control loops: (1) the water level inside the boiler is controlled by manipulating the pump throughput; and (2) the output flow of vapor is controlled by manipulating the heater power. The two PID have limited output, anti-windup compensation and setpoint weightings. Each PID has the following interactive parameters: proportional gain (K_p), integral time constant (T_i), derivative time constant (T_d), setpoint weight for the proportional term (w_p), setpoint weight for the derivative term (w_d), anti wind-up compensator constant (N_i), derivative filter parameter (N_d), lower limit for the output (y_{min}) and upper limit for the output (y_{max}).

6.8.2 Virtual-lab view

The Modelica description of the virtual-lab view is shown in Figure 6.6. It automatically generates the Java code of the interactive graphic interface shown in Figure 6.7. 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 three components hosted inside it: mainFrame of MainFrame class and dialog and

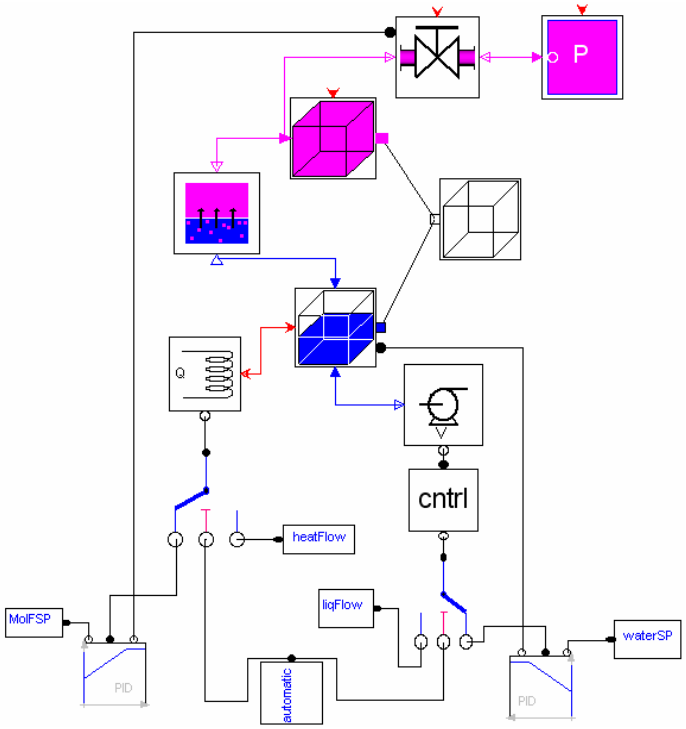


Figure 6.5: Diagram of the boiler model.

dialog1 of Dialog class. The components mainFrame and dialog generate the two windows shown in Figure 6.7.

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, InfoButton, PauseButton and Slider classes. The two radio-buttons allow the user to select the control

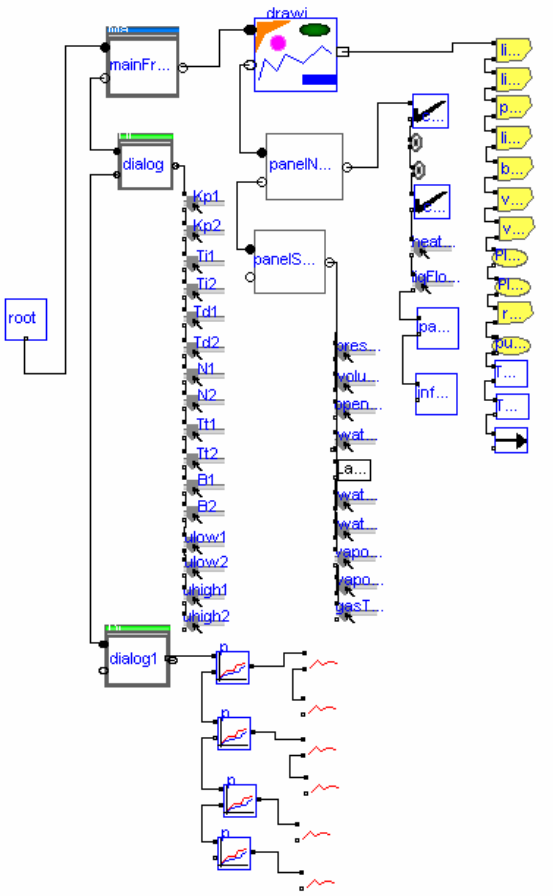


Figure 6.6: Diagram of the Modelica description of the view.

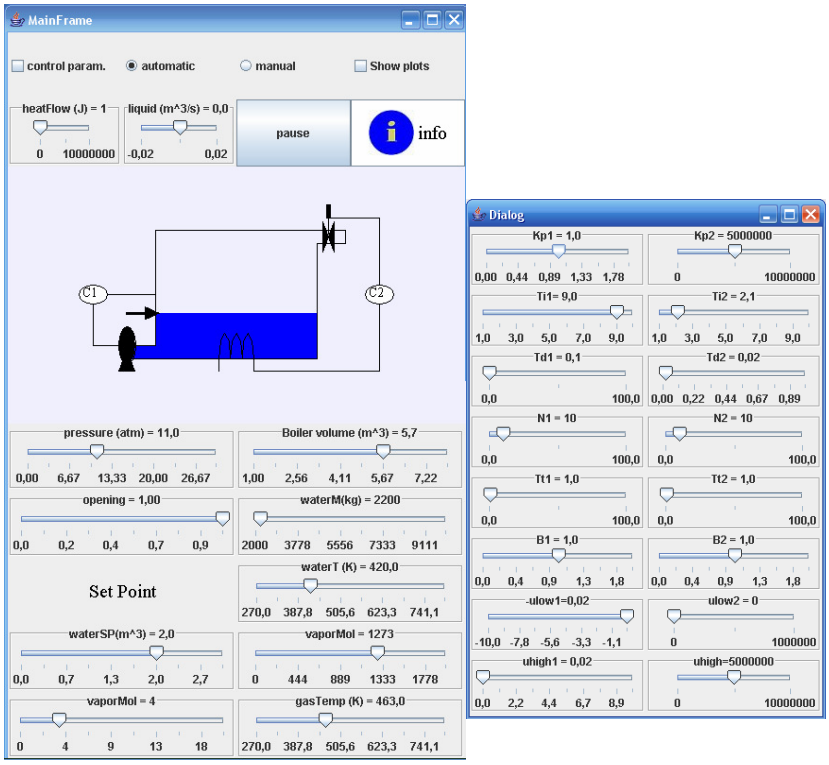


Figure 6.7: View of the boiler virtual-lab

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. The button of the InfoButton and PauseButton classes allows the user, respectively, to pause and resume the simulation and to display a window with the information about the virtual-lab (see figure 6.8).

– 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 Slider class. These sliders allow the user to change the parameter values of the two PID controllers.

The dialog1 container generates the graphic interface shown in Figure 6.9. This container hosts components of PlottingPanel class which contain drawables of Trail class. These drawables generate traces that show the time evolution of some relevant system variables (see Figure 6.9).

6.8.3 Virtual-lab set up and launch

The virtual-lab description is obtained as discussed in Section 6.3. It is translated using Dymola and executed. Then, the jar file containing the Java code of the virtual-lab view is automatically generated and executed. Then, the virtual-lab view is displayed (see Figure 6.7).

The dynamic response of the boiler to a step change in the output pressure is shown in Figure 6.9. This change has been interactively performed by the virtual-lab user at the simulated time 243 s. The boiler is operating in automatic control mode. The following four plots are shown in Figure 6.9:

1. Actual value of the vapor flow and its setpoint.

2. Heat generated by the heater.

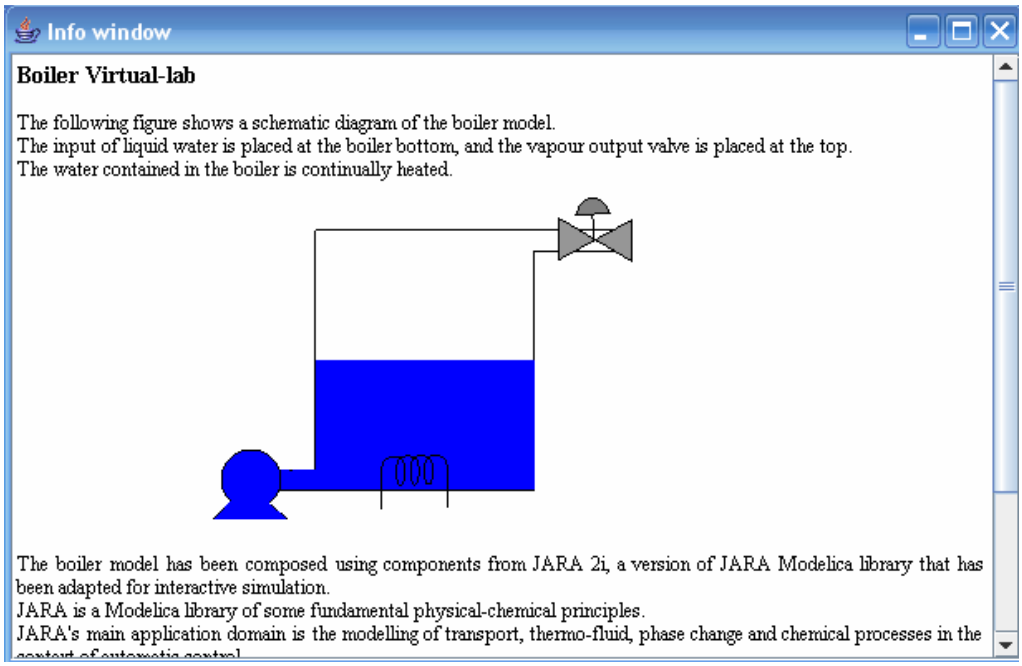


Figure 6.8: Introduction of the boiler virtual-lab.

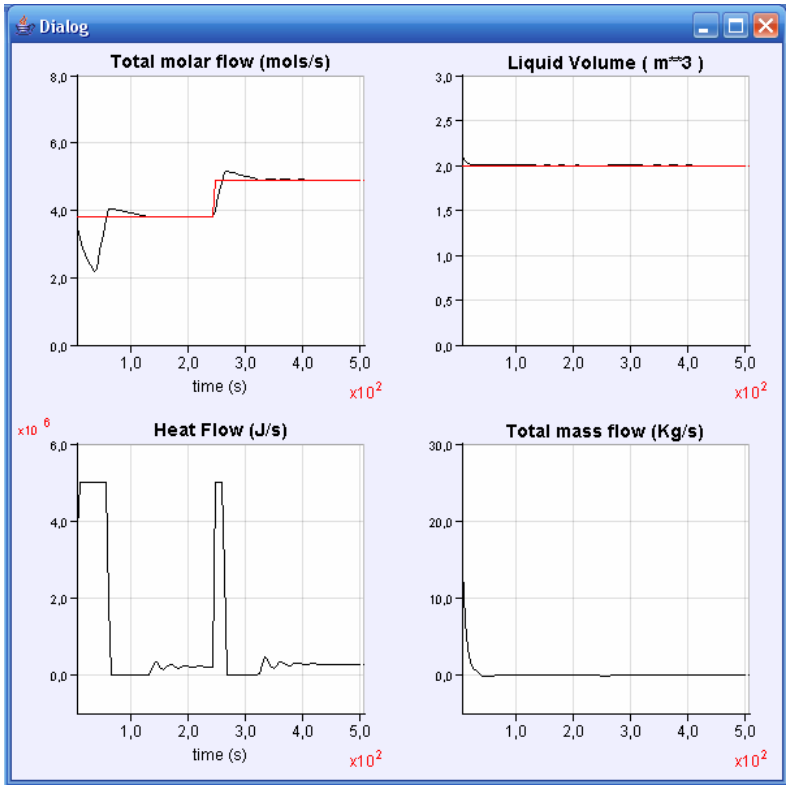


Figure 6.9: Time evolution of some selected variables of the boiler virtual-lab.

3. Actual value of the water volume contained inside the boiler, and its setpoint value.

4. Liquid flow rate generated by the pump.

6.9 Case study II: virtual-lab of a heat-exchanger

This virtual-lab illustrates the dynamic behavior of a double-pipe heat exchanger. The model of this virtual-lab has been built using the JARA 2i library. This model was discussed in Section 2.5.5.

6.9.1 Virtual-lab view

The Modelica description of the virtual-lab view and the Java view generated are shown in Figures 6.10 and 6.11, respectively. The relationship between the Modelica description and the corresponding graphic interface is briefly explained next.

The root component has two components hosted inside it: MF of MainFrame class and dialog of Dialog class. The components MF and dialog generate the two windows shown in Figure 6.11.

The MF 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: DP (of DrawingPanel class), panelN and panelS (of Panel class).

- DP is placed in the center of the MF. This component contains the animated diagram of the longitudinal section of the heat-exchanger. This diagram is composed of *drawable elements* of Polygon and PolygonSet classes. Each liquid control volume has been represented by a rectangular polygon whose filling color depends on the temperature of the liquid inside the control volume. Analogously, each gas control volume has been represented by a rectangular polygon whose temperature depends on the temperature of the gas inside

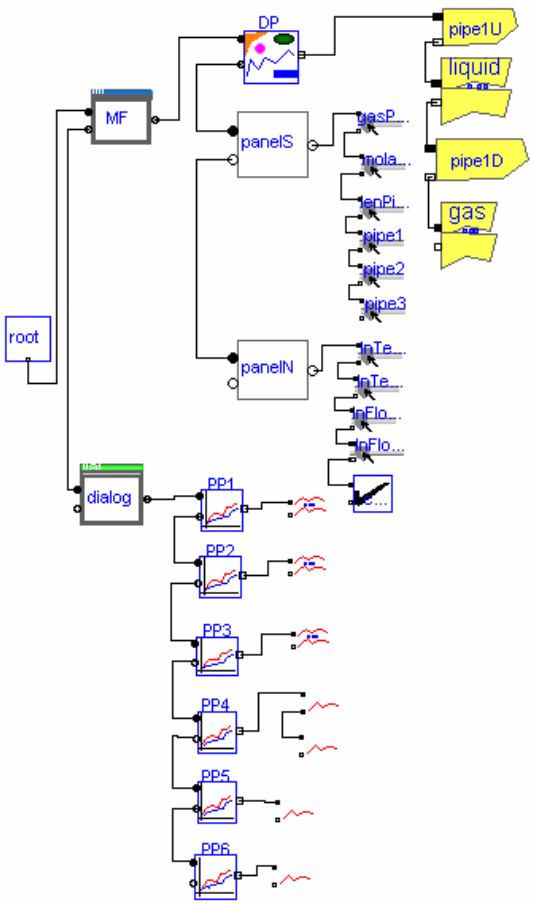


Figure 6.10: Modelica description of the heat-exchanger virtual-lab view.

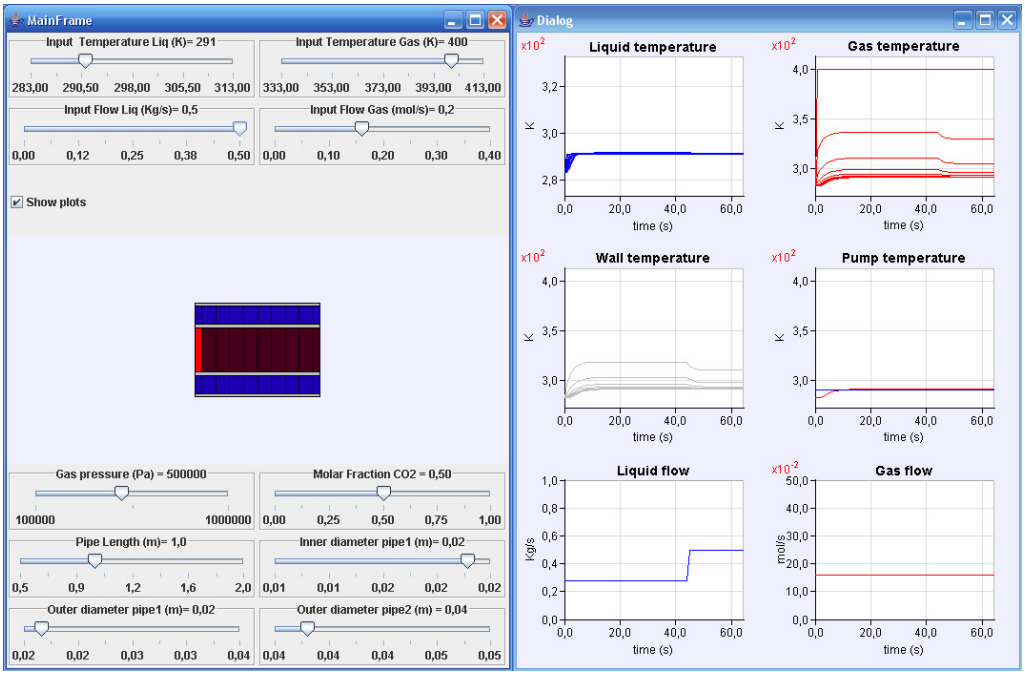


Figure 6.11: View of the heat-exchanger virtual-lab.

the control volume. Color changes from blue (lower temperatures) to red (higher temperatures).

– panelN hosts interactive controls of Slider and CheckBox classes. The sliders allow the user to change the pump input temperature, and flow of liquid and gas. The checkbox allows the user to show and hide the "*Dialog*" window.

– panelSouth hosts interactive controls of Slider class. These sliders allow the user to perform interactive changes in the value of the pipe length, inner and outer diameter of the inner pipe, outer diameter of the outer pipe, gas pressure and molar fraction of CO_2.

The dialog container generates the right window shown in Figure 6.11. This container hosts components of PlottingPanel class, which contain drawables of Trail and TrailSet classes. These drawables generate traces that show the time evolution of some relevant system variables.

6.9.2 Virtual-lab set up and launch

The virtual-lab description is obtained as discussed in Section 6.3. It is translated using Dymola and executed. Then, the jar file containing the Java code of the virtual-lab view is automatically generated and executed, and the virtual-lab view is displayed (see Figure 6.11).

The response to a step change (from 0.3 to 0.5 kg/s) in the liquid flow, performed interactively at $time = 44$ s, is shown in Figure 6.11. The following six plots are shown in the right window of the Figure 6.11:

1. Temperature of each liquid control volume.
2. Temperature of each gas control volume.
3. Temperature of each solid control volume corresponding to the inner pipe wall.
4. Temperature of the liquid flow generated by the pump.
5. Liquid flow rate generated by the pump.

6. Gas flow rate generated by the pump.

6.10 Case study III: virtual-lab of a washing machine

The implementation of a virtual-lab for testing designs of drum-type washing machines is discussed. It is applied to the analysis of an industrial washing machine (120 Kg load capacity) manufactured by Fagor Industrial. The work presented in this section is the result of a collaboration with the Mechanical Engineering Department of the IKERLAN Technological Research Center (Mondragón, Spain). The physical model of the drum-type washing machine and an important part of the virtual-lab view design has been developed by the IKERLAN engineers. The adaptation of the physical model to be suitable for interactive simulation and the implementation of the virtual-lab view have been part of this thesis work.

The virtual-lab supports interactive changes in the position and properties of the springs and the dampers, the properties of the inner and outer drums, and the mass and position of the load. Simulation results are in good agreement with the experimental data. The virtual-lab has demonstrated to be a valuable design and analysis tool, allowing the user:

- To get insight into the system behavior.
- To tune the system parameters in order to improve the dynamic behavior.
- To simulate special events, such as a component breakage.

6.10.1 Washing machine dynamic analysis

Drum-type washing machines are widely used in Europe. They are composed of an inner drum that rotates inside an outer drum, with a horizontal axis, making the clothes tumble upward and downward during washing cycle (see Figure 6.12). During the drying cycle, clothes are subjected to both the gravity force (g) and the centrifugal force, generated by the inner drum rotational speed.

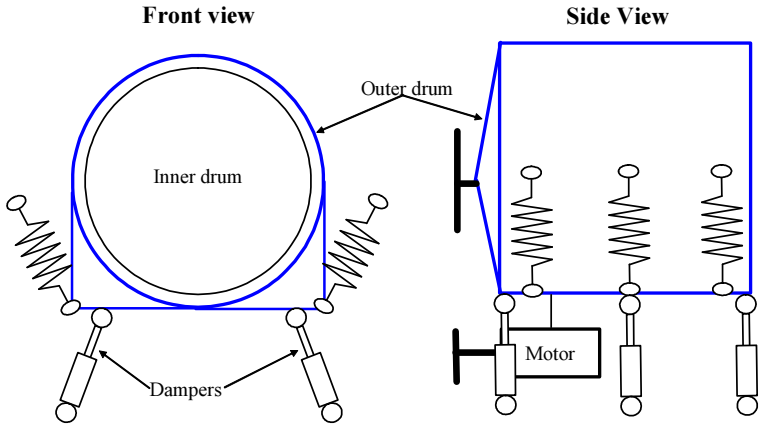


Figure 6.12: Schematic dynamic model of the washing machine.

When the centrifugal force is bigger than "g", the clothes tend to stick to the inner drum wall. In some cases, it results in a non homogeneous distribution of the clothes' mass around the periphery of the inner drum. This is mainly due to the different composition of the tissues. Imbalance occurs when clothes' center of mass does not coincide with the inner drum rotation axis, and it induces vibration to the outer drum.

In order to reduce the vibrations transmitted to the floor, the outer drum is suspended with springs. The forces transmitted to the frame (floor) can be drastically reduced if the resulting natural frequency (spring-drum) is very low. On the other hand, suspended drum movements can become uncontrollable when passing through the natural frequency and at low rotational speeds, which can cause collisions against the frame. Friction dampers are normally used to limit these movements.

Suspended drum movement depends on many factors. For example, the suspended mass inertia, the spring and damper positions and characteristics, the unbalanced mass value and location, and the spinning speed profile. All these parameters must be tuned for each new design, in order to minimize the drum displacements and the forces transmitted to the frame.

Accurate models of the drum dynamic, including unbalance load effects, can not be derived analytically due to the complexity of the dynamic behavior, influenced by those parameters and their interactions. This limitation is even more

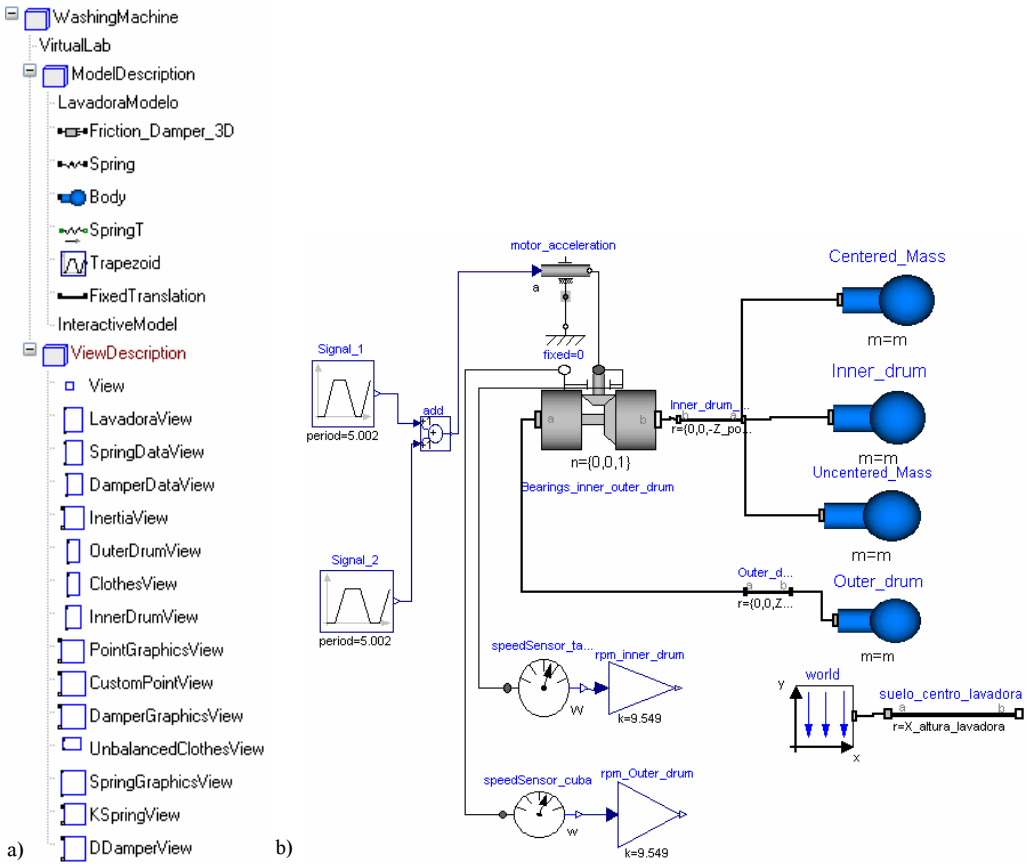


Figure 6.13: a) WashingMachine library; and b) Modelica diagram of the washing machine physical model.

evident when analyzing big-size washing machines (40 to 120 kg load capacity), which are suspended by several couples of springs and dampers. The dynamic behavior of the suspended drum can be successfully analyzed using rigid-body dynamic modeling and computer simulation.

6.10.2 Multibody model

The Modelica classes required to describe the washing machine virtual-lab are contained in the *WashingMachine* library (see Figure 6.13a). This library is composed of the following two packages: *ModelDescription* and *ViewDescription*. These packages contain the classes required to describe the model and the view of the virtual-lab, respectively.

The Modelica diagram of the washing machine is shown in Figure 6.13b. It has been composed using models contained in the *MultiBody* library (Otter et al.

2003), which is one of the Modelica Standard libraries (*Modelica* 2007). An application of *MultiBody* library to the modeling of a household washing machine is described in (Ferreti & Schiavo 2006).

All the bodies, except the springs, have been considered rigid. The suspended drum is composed of the following four bodies: outer drum, inner drum, centered and un-centered mass. These masses are attached to the inner drum. One rotational degree of freedom (DoF) is allowed between the inner and the outer drums.

The suspended drum has six DoF. Its dynamic behavior is governed by the forces generated by the mass of the uncentered clothes, the gravity, and the forces exerted by three pairs of springs and dampers. The springs and dampers are modeled as ideal elements (i.e., the force is proportional to the relative displacement or speed, respectively). Additionally, an external mass-free frame is considered. The springs and dampers are attached to this frame. This approach allows the computation of the floor reaction forces.

The model has been adapted applying the methodology discussed in Section 4.2 to allow interactive changes in the position and properties of the springs and the dampers, the properties of the inner and outer drums, and the mass and position of the load. The model is intended to be used for tuning the value of these parameters, in order to improve the washing machine dynamic behavior. The evaluation of the suspended system displacement is accomplished for the following two critical test conditions: spinning start up and spinning at maximum speed. The dynamic behavior analysis is based on the following two key magnitudes: (1) the displacement of the suspended system with respect to the external frame; and (2) the forces transmitted to the floor. These forces can cause vibrations and relative displacements of the frame.

6.10.3 Virtual-lab view

The Modelica description of the virtual-lab view has been developed modularly, by extending and connecting the required graphic components of the *VirtualLabBuilder* library. The diagram of the Modelica class describing the virtual-lab

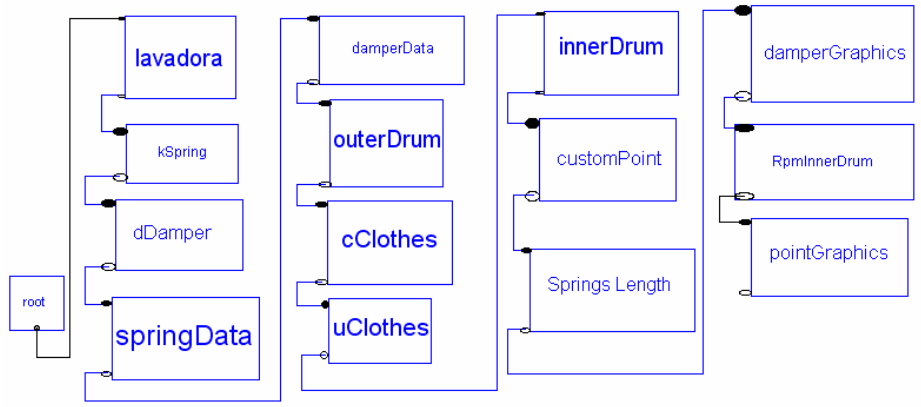


Figure 6.14: Modelica description of the washing-machine virtual-lab view.

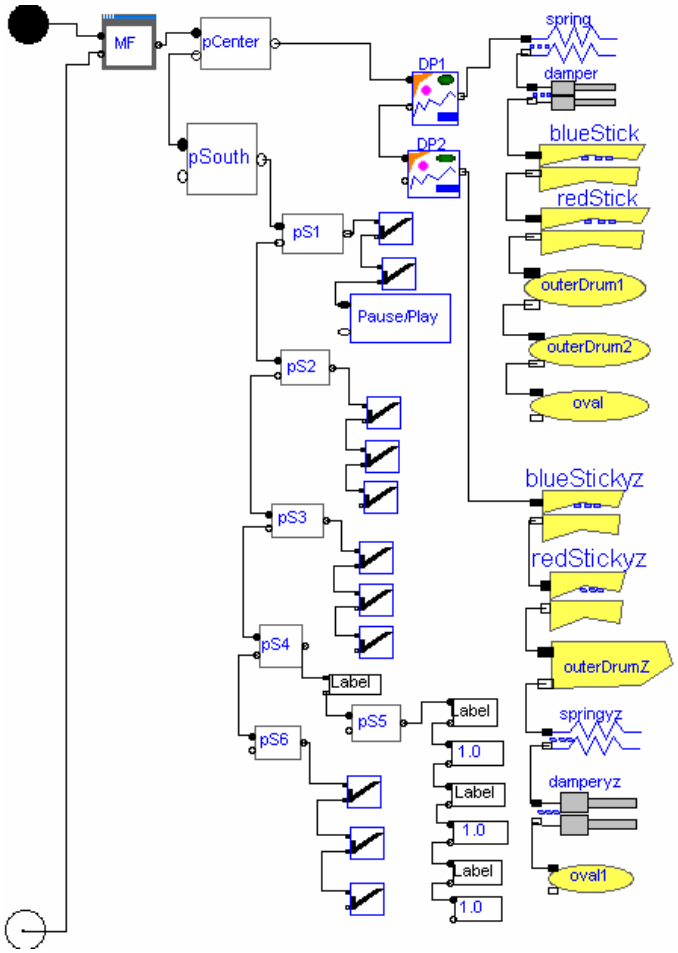
view is shown in Figure 6.14. The view contains one main window and 15 dialog windows. Each window of the virtual-lab view is described by a class. The classes describing the main window and the dialog windows are briefly described below.

The diagram of the Modelica class describing the main window is shown in Figure 6.15a. The component MF - of MainFrame class - generates the window shown in Figure 6.15b. The MF 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). It has two components hosted inside it: pCenter and pSouth, both of Panel class.

- pCenter is placed in the center of the MF. This component contain the two following containers of the DrawingPanel class: DP1 and DP2. These two components contain, respectively, the animated diagram of the frontal and lateral animated diagrams of the washing machine. These two diagrams are composed of several *drawable elements* of Polygon, PolygonSet, Oval, DamperSet and SpringSet classes.

- pSouth hosts several interactive controls of PauseButton, CheckBox, Label and NumberField classes. Checkboxes allows the user to show and to hide the dialog windows. The button allows to pause and resume the simulation. The spatial coordinates of system points are set using NumberField class components.

a)



b)

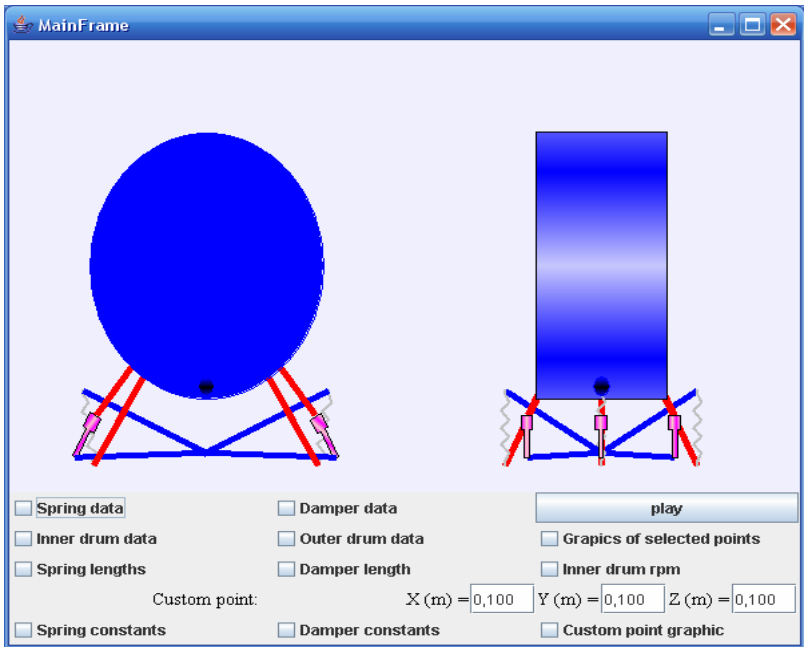


Figure 6.15: Main window of the washing machine virtual-lab: a) Modelica diagram; and b) Java view.

There are two types of dialog windows: (1) the windows containing plots that display the time evolution of some model variables; and (2) the windows containing interactive controls that allow the user to perform interactive changes in the model variables.

The following windows contain the interactive controls (see Figure 6.16):

- "*Spring Data*" window allows changing the position of the springs extremities in relation to the frame and the outer drum.
- "*Damper Data*" window allows changing the position of the dampers extremities in relation to the frame and the outer drum.
- "*Inner Drum*" window allows changing the value of relevant properties of the inner drum, including radius, mass, length, center of gravity (C.O.G) position, center position and sheave position. Additionally, this window contain checkboxes that allow the user to show and hide three dialog windows. These three windows contain interactive control elements that allow changing the C.O.G. mass and position of the centered and unbalanced load and the inertia matrix of the inner drum. The Modelica diagram associated to this window and the graphic interface generated are shown, respectively, in Figures 6.17a and 6.17b.
- "*Outer Drum*" window allows changing the value of the properties of the outer drum (i.e., radius, mass, inertia and position of its C.O.G).
- "*Spring constant*" window allows changing the value of the spring constants.
- "*Damper constant*" window allows changing the value of the damper constants.

The virtual-lab contains five plot windows displaying the time-evolution of the following variables:

- Damper lengths. The Modelica diagram associated to the window displaying the time evolution of the damper lengths and the graphic interface generated are shown, respectively, in Figures 6.18a and 6.18b. The Modelica

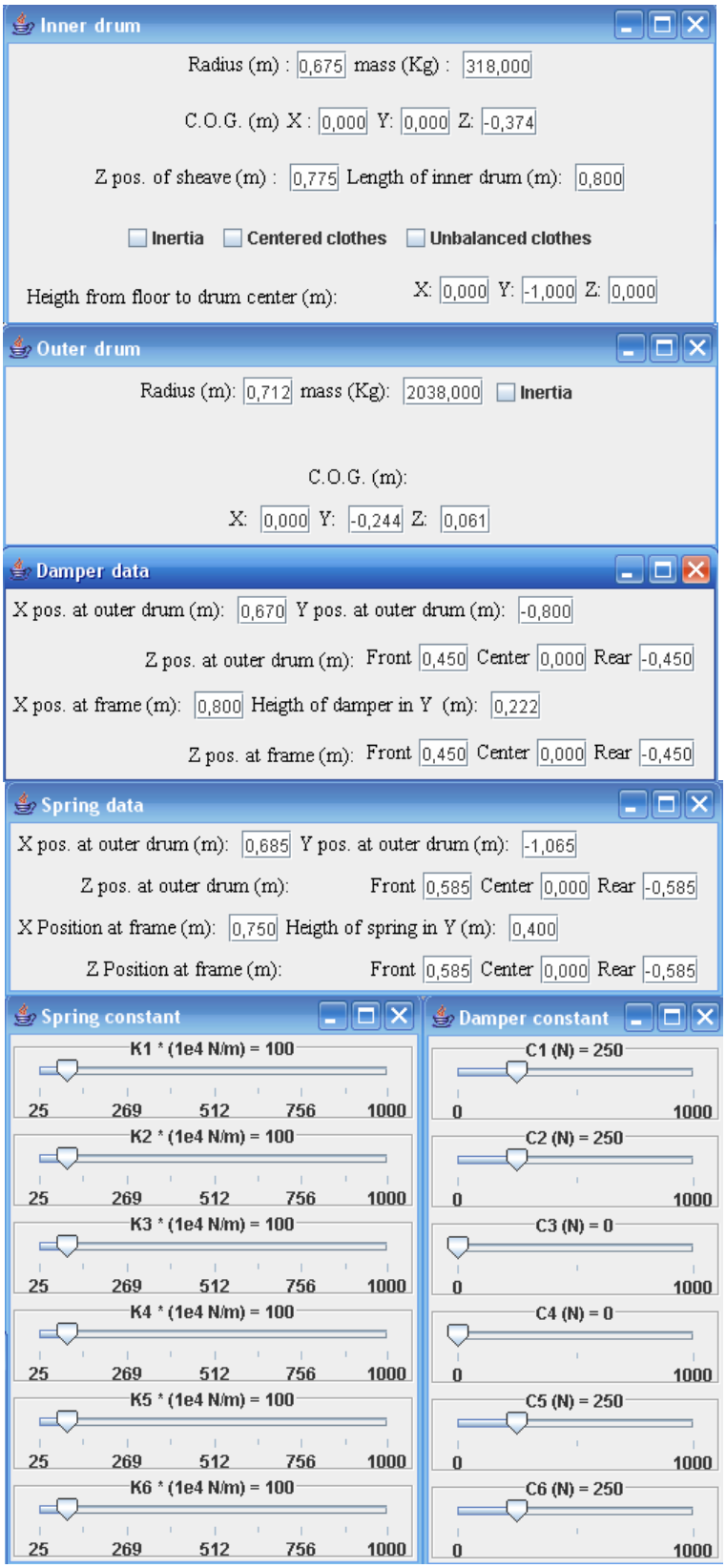


Figure 6.16: Windows "Spring Data", "Damper Data", "Inner Drum", "Outer Drum", "Spring constant" and "Damper constant" of the washing machine virtual-lab.

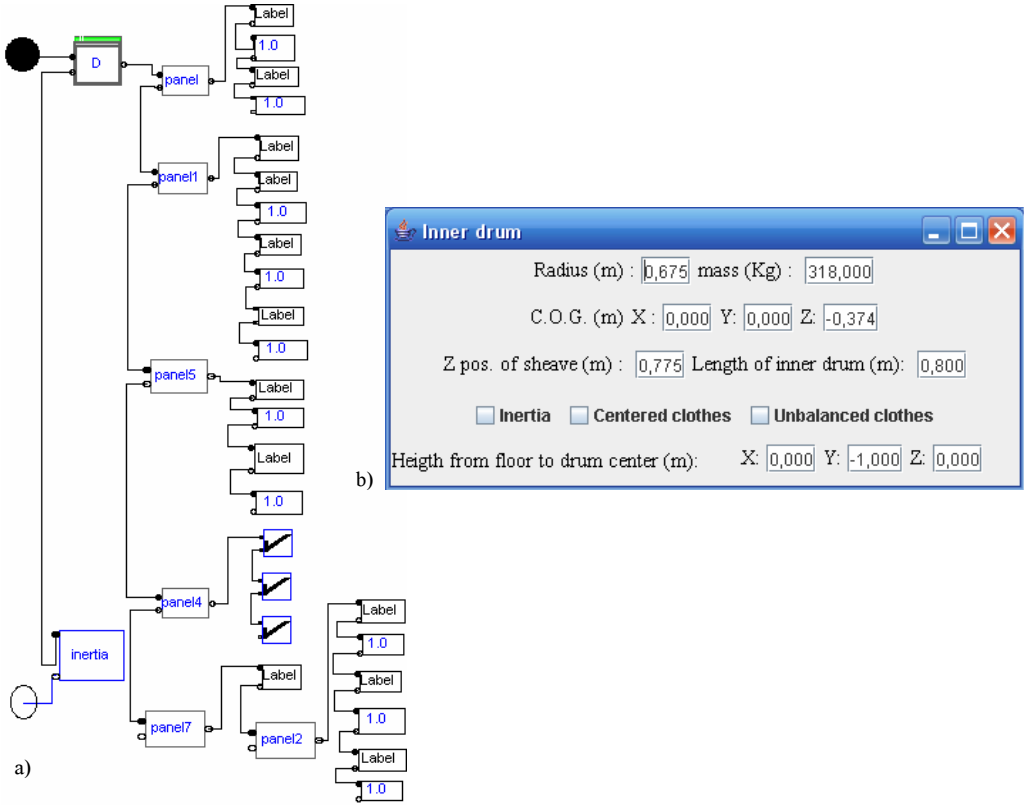


Figure 6.17: "Inner drum" window of the washing machine virtual-lab view: a) Modelica diagram; and b) Java view.

diagram describing this window contain components of the PlottingPanel and Trail classes.

- Spring lengths.
- Position of a system point, which can be interactively chosen by the virtual-lab user.
- Position of certain relevant points of the system.
- Rotational speed of the inner drum.

6.10.4 Virtual-lab set up and launch

The virtual-lab description is obtained as discussed in Section 6.3. It is translated using Dymola and executed. Then, the jar file containing the Java code of the

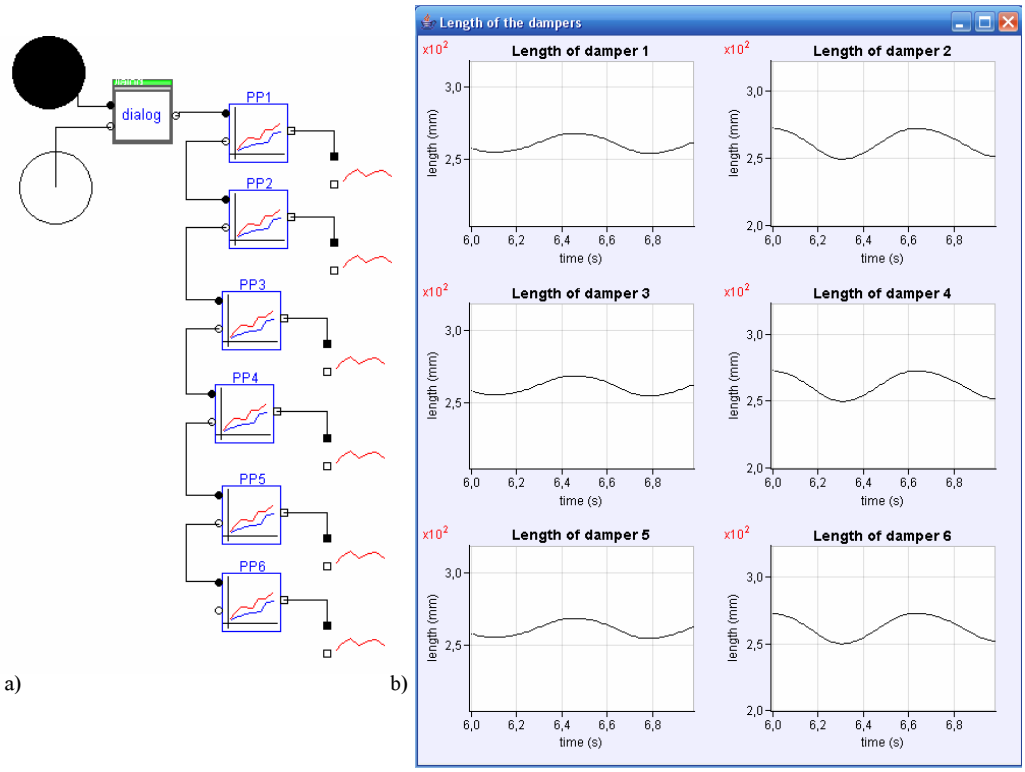


Figure 6.18: "Inner drum" window of the washing machine virtual-lab view: a) Modelica diagram; and b) Java view.

virtual-lab view is automatically generated and executed and the virtual-lab view is displayed (see Figure 6.15b).

The time evolution of the system point whose position can be interactively set by the virtual-lab user, the spring and the damper lengths are shown respectively in Figures 6.19, 6.20 and 6.21. The system specifications are the ones displayed by the windows shown in Figure 6.16. The speed profile of the inner drum is shown in Figure 6.22.

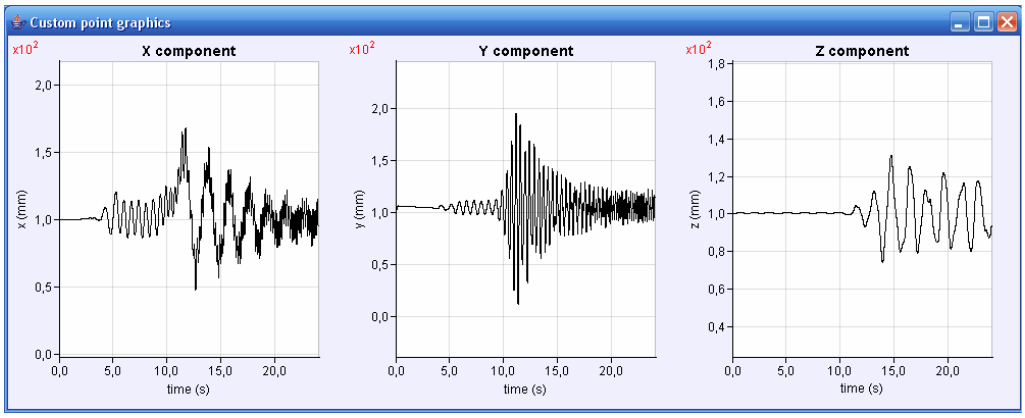


Figure 6.19: Time evolution of the point whose position can be selected by the virtual-lab user.

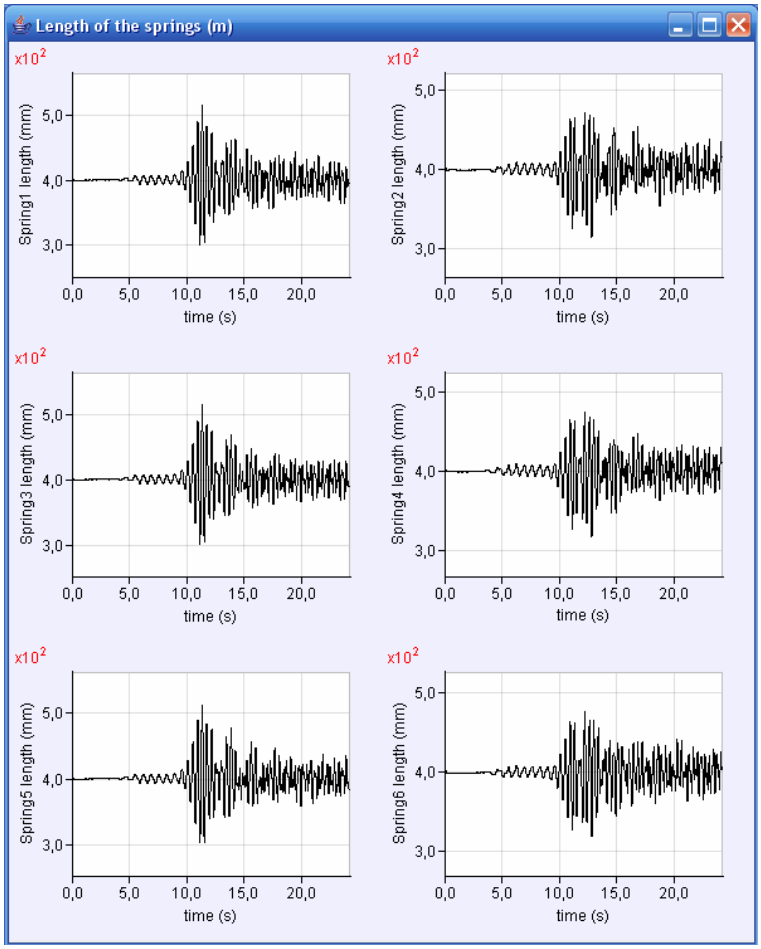


Figure 6.20: Time evolution of the spring lengths.

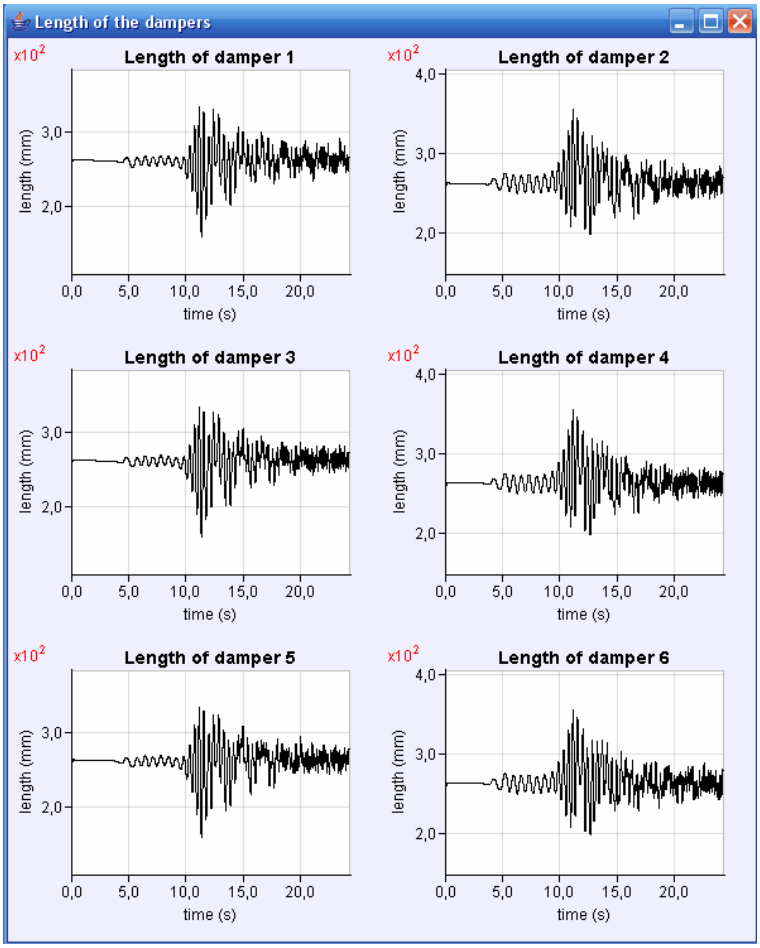


Figure 6.21: Time evolution of the damper lengths.

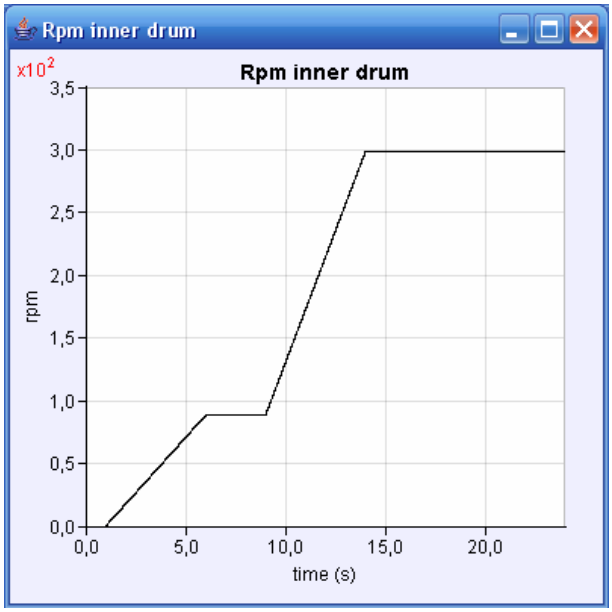


Figure 6.22: Speed profile of the inner drum.

6.11 Conclusions

This chapter has provided the essential information to build a virtual-lab completely described in Modelica language using the *VirtualLabBuilder* Modelica library. For that purpose, the following topics have been discussed:

- The procedure proposed to build a virtual-lab using the *VirtualLabBuilder* library.
- The architecture of *VirtualLabBuilder* library, the interactive graphic elements included in the library and the connection rules that the library user has to follow to build the view description.

Additionally, the following three case studies have been discussed:

- The industrial boiler and the heat-exchanger virtual-labs, useful as educational tools.
- The drum-type washing machine virtual-lab, a useful design aid.

7

VirtualLabBuilder Modelica Library - Developer's Perspective

7.1 Introduction

Design and implementation details useful for the developer of the *VirtualLabBuilder* library are provided in this chapter. In particular, the following topics are addressed:

- The procedure to implement new interactive graphic elements.
- The relationship between the structure of the view description in Modelica and the Java code generation.
- The communication between the model and the Java view.

7.2 Structure of the src package

The *VirtualLabBuilder* packages containing the classes to be used by the virtual-lab developers were described in Section 6.4. The structure of the src package is described below (see Figure 7.1):

VLabModel package includes the PartialView, Root and VirtualLab classes. PartialView and VirtualLab classes are inherited from the classes with the same name contained in the VirtualLabBuilder.VLabModel package.

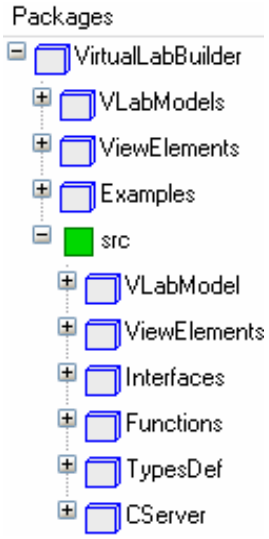


Figure 7.1: Structure of the src package.

ViewElements package includes the Containers, Drawables, InteractiveControls and BasicElements packages. They contain classes describing the interactive graphic elements and their base classes. The library developer has to extend these base classes to implement new interactive graphic elements. The procedure to implement new interactive graphic elements will be discussed in Section 7.4.

Interfaces package includes the connectors and interfaces of the interactive graphic elements. They will be discussed in Section 7.3.

Functions package includes:

- Modelica functions embedding external C-functions, which are Java code generators.
- *processingFile* Modelica function. It will be described in Section 7.5.
- Some other Modelica functions, which are used by the interactive graphic elements.

TypesDef package includes type declarations. They are intended to be used for defining some properties of the interactive graphic elements, such as the color, the layout, etc.

Name	Icon	Variables	Interfaces
ParentL		nodeReference BorderLayout	IContainers IContainerDrawables IViewElements
ChildL		nodeRefence BorderLayout	IContainers IContainerDrawables IViewElements
Parent		nodeReference	IDrawables
Child		nodeReference	IContainerDrawables IDrawables

Figure 7.2: Connectors included in the *VirtualLabBuilder* library.

CServer package includes Modelica functions encapsulating external C-functions. The goal of these external C-functions is to implement the communication between the executable C-file generated by Dymola and the virtual-lab GUI (i.e., the Java program automatically generated during the initialization process).

7.3 Interface of the interactive graphic elements

The Interfaces package includes the connectors and interfaces of the interactive graphic elements. The following four classes of interface have been implemented: IContainer, IContainerDrawables, IDrawables and IViewElements. The connectors and interfaces defined in the Interfaces package are discussed below.

7.3.1 Connectors

The following four types of connectors have been defined (see Figure 7.2):

- The **ParentL** and **ChildL** connectors have the following two variables:
  - The *nodeReference* variable is an integer number that identifies univocally each one of the interactive graphic objects that compose the class describing the virtual-lab view.

· *BorderLayout* is a boolean variable whose value is *true* if the component's layout policy is *BorderLayout*.

– The **Parent** and **Child** connectors have only one variable: *nodeReference*. The meaning of this variable is the same as in the ParentL and ChildL connectors.

7.3.2 IContainer interface

The IContainer interface is inherited from classes describing containers that don't host drawable elements. It contains:

– Two "*left*" connectors: (1) pLLeft, of ParentL class; and (2) cLLeft, of ChildL class. The interface contains equations to transmit the value of the pLLeft's variables to the cLLeft's variables (see Modelica Code 7.1).

– An integer variable, called *num*. Its value is obtained during the model initialization process. This value identifies univocally each one of the interactive graphic elements composing the Modelica view description. The computation of the value of the *num* variable is discussed in Section 7.5. The *num* variable is also defined in the other three types of interfaces.

– A String parameter, named *LayoutPolicy*. The value of this parameter sets the layout policy of the container (i.e. $BorderLayout()$, $HorizontalBox()$, $VerticalBox()$, *GridLayout* or $FlowLayout$).

– One "*right*" connector of ChildL class (cLRight). The connector variables are calculated from *num* and *LayoutPolicy* (see Modelica Code 7.1).

7.3.3 IContainerDrawables interface

The IContainerDrawable interface is inherited from classes describing containers that only host drawable elements. It contains:

– Two "*left*" connectors: (1) pLLeft, of ParentL class; and (2) cLLeft, of ChildL class. The interface contains equations to transmit the value of the pLLeft's variables to the cLLeft's variables (see Modelica Code 7.2).

```
partial model IContainer
  import Modelica.Utilities.*;
  Interfaces.ParentL pLLeft annotation (extent=[-100,18; -80,38]);
  Interfaces.ChildL cLRight annotation (extent=[80,-10; 100,10]);
  Interfaces.ChildL cLLeft annotation (extent=[-100,-40; -80,-20]);
  parameter TypesDef.LayoutPolicy LayoutPolicy = "BorderLayout()"
                                  "Layout policy of the component";
protected
  Integer num "Number identifying the component
               in the virtual-lab view description";
initial algorithm
  cLRight.nodeReference := num;
  cLRight.borderLayout := if (Strings.compare(LayoutPolicy, "BorderLayout()")
                              == Types.Compare.Equal) then true else false;
  cLLeft.nodeReference := pLLeft.nodeReference;
  cLLeft.borderLayout := pLLeft.borderLayout;
equation
  when false then
    num = pre(num);
    cLRight.nodeReference = pre(cLRight.nodeReference);
    cLRight.borderLayout = pre(cLRight.borderLayout);
    cLLeft.nodeReference = pre(cLLeft.nodeReference);
    cLLeft.borderLayout = pre(cLLeft.borderLayout);
  end when;
  annotation (Diagram);
end IContainer;
```

Modelica Code 7.1: Partial model `IContainer`.

```
partial model IContainerDrawables
  import Modelica.Utilities.*;
  Interfaces.ParentL pLLeft annotation (extent=[-100,40; -80,60]);
  Interfaces.ChildL cLLeft annotation (extent=[-100,-40; -80,-20]);
  Interfaces.Child cRight annotation (extent=[80,-10; 100,10]);
protected
  Integer num "Number identifying the component
               in the virtual-lab view description";
initial algorithm
  cRight.nodeReference := num;
  cLLeft.nodeReference := pLLeft.nodeReference;
  cLLeft.borderLayout := pLLeft.borderLayout;
equation
  when false then
    num = pre(num);
    cRight.nodeReference = pre(cRight.nodeReference);
    cLLeft.nodeReference = pre(cLLeft.nodeReference);
    cLLeft.borderLayout = pre(cLLeft.borderLayout);
  end when;
end IContainerDrawables;
```

Modelica Code 7.2: Partial model `IContainerDrawables`.

```
partial model IDrawable
  import Modelica.Utilities.*;
  Interfaces.Parent pLeft annotation (extent=[-100,40; -80,60]);
  Interfaces.Child cLeft annotation (extent=[-100,-40; -80,-20]);
protected
  Integer num "Number identifying the component
               in the virtual-lab view description";
  Integer dummy;
initial algorithm
  cLeft.nodeReference := pLeft.nodeReference;
  dummy :=num;
equation
  when false then
    num = pre(num);
    dummy = pre(dummy);
    cLeft.nodeReference = pre(cLeft.nodeReference);
  end when;
end IDrawable;
```

Modelica Code 7.3: Partial model `IDrawable`.

- The *num* variable.
- One "*right*" connector of Child class (cRight). The connector variable is equal to the *num* variable (see Modelica Code 7.2).

7.3.4 IDrawable interface

The IDrawable interface is inherited from classes describing drawable elements. It contains:

- Two "*left*" connectors: (1) pLeft, of Parent class; and (2) cLeft, of Child class. The interface contains equations to transmit the value of the pLeft's variables to the cLeft's variables (see Modelica Code 7.3).
- The *num* variable.

7.3.5 IViewElement interface

The IViewElement interface is inherited from classes describing basic and interactive elements. It contains:

```
model IViewElement
  import Modelica.Utilities.*;
  Interfaces.ParentL pLLeft annotation (extent=[-100,40; -80,60]);
  Interfaces.ChildL cLLeft annotation (extent=[-100,-40; -80,-20]);
protected
  Integer num;
  Integer dummy;
initial algorithm
  cLLeft.nodeReference := pLLeft.nodeReference;
  cLLeft.borderLayout := pLLeft.borderLayout;
  dummy := num;
equation
  when false then
    num = pre(num);
    dummy = pre(dummy);
    cLLeft.nodeReference = pre(cLLeft.nodeReference);
    cLLeft.borderLayout = pre(cLLeft.borderLayout);
  end when;
end IViewElement;
```

Modelica Code 7.4: Model `IViewElement`.

- Two "*left*" connectors: (1) pLLeft, of ParentL class; and (2) cLLeft, of ChildL class. The interface contains equations to transmit the value of the pLeft's variables to the cLeft's variables (see Modelica Code 7.4).
- The *num* variable.

7.4 Implementing new interactive graphic elements

Each interactive element of *VirtualLabBuilder* has associated the following three elements:

1. A Modelica class. Details about the Modelica class are discussed in the rest of this section.
2. A Java class. All the Java classes describing interactive components are packed in a jar file named *graphics.jar*. Some of these classes are based on (*Open Source Physics* 2007).

3. A Modelica function encapsulating a C function. Its objective is writing to a file the code required to create an instance of the Java class describing the interactive element.

7.4.1 The Modelica class

The structure of the *VirtualLabBuilder* Modelica classes describing interactive elements is as follows:

- The class inherits from a base class. The Containers, Drawables, InteractiveElements and BasicElements packages contain the base classes required to create new interactive graphic elements. These base classes are discussed in Section 7.4.2.
- The declaration of the parameters needed to set the interactive element properties.
- The section "*initial algorithm*", which has to contain the code required to:
  - Calculate the value of the *num* variable. This is accomplished by executing the function *processingFile*.
  - Call the Modelica function. This Modelica function calls a C function which write to a file the Java code.

7.4.2 Base classes

The base classes included in the Containers, Drawables, InteractiveElements and BasicElements packages are discussed in this section.

The relationship among the interfaces, the base classes, and the classes describing the interactive graphic elements are shown in Figures 7.3, 7.4 and 7.5. The following symbol terminology has been used to make these representations:

- Classes are placed inside rectangles.
- Partial class are placed inside rectangles with dashed line borders.

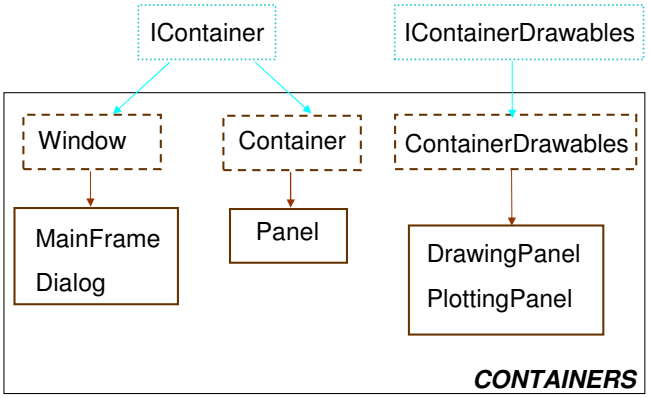


Figure 7.3: Classes included in the Containers package.

– An arrow going from a rectangle A to a rectangle B indicates that the classes within the rectangle B inherit from the classes within the rectangle A.

Containers package

The Containers package includes the following three base classes (see Figure 7.3): Window, Container and ContainerDrawables. These three classes are described below:

– Window class is inherited from classes describing interactive graphic elements that create windows. This class inherits from the IContainer class. It contains the declaration of the parameters needed to specify the title of the window, its width and position, and the number of row and columns if the *GridLayout* policy is selected.

– Container class is inherited from classes describing interactive graphic elements that create panels which can't host drawables elements. This class inherits from the IContainer class.

– ContainerDrawables class is inherited from classes describing interactive graphic elements that create panels which can only host drawables elements. This class inherits from the IContainerDrawables class.

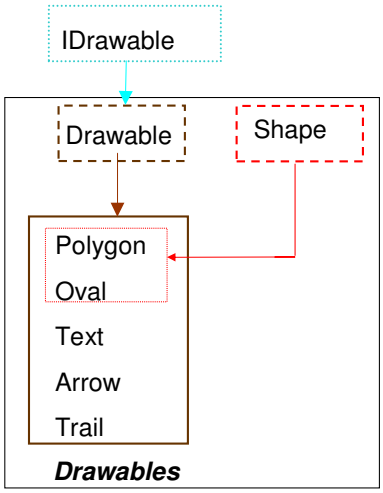


Figure 7.4: Classes included in the Drawables package.

Drawables package

The Drawables package includes the following two base classes (see Figure 7.4): Drawable and Shape. These two classes are described below:

- Drawable class is inherited from classes describing drawables elements. This class inherits from the IDrawable class. It includes the code required to:
  - Send data from the drawable element to the Java program (automatically generated during the model initialization process).
  - Finish the simulation when the main window of the Java program is closed.

  These two communication tasks will be discussed in Section 7.6.

- Shape class is inherited from classes describing 2-D drawables with shape (i.e., Polygon and Oval). This class includes the following parameters in order to describe the color properties of the drawable element:

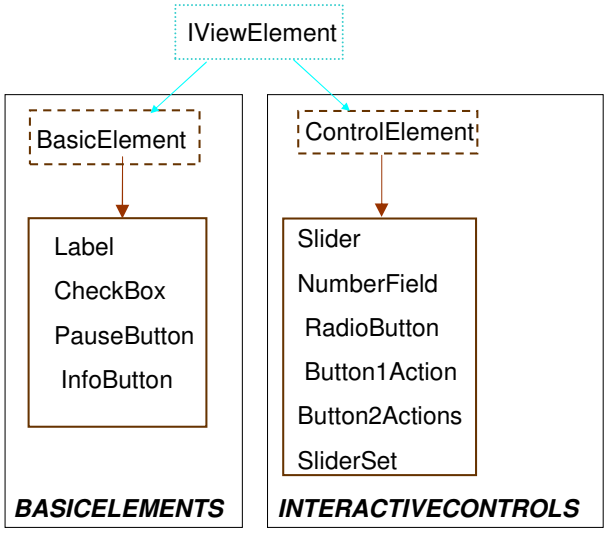


Figure 7.5: Classes included in the InteractiveElements and BasicElements packages.

- *filled* "*True*" if the polygon is filled, "*False*" otherwise.
- *lineColorp*[4] vector describing the line color of the drawable.
- *fillColorp*[4] vector describing the color used to fill the component.
- *intLineColor* 1 if the line color changes in time, 0 otherwise.
- *intFillColor* 1 if the filling color changes in time, 0 otherwise.

This class includes the declaration of the following two variables: $lineColor[4]$ and $fillColor[4]$. If the $intlineColor$ / $intfillColor$ parameter is 0, then the value of $lineColor[4]$ / $fillColor[4]$ is set to the value of the $lineColorp[4]$ / $fillColorp[4]$ parameters. Otherwise, the value of these variables has to be set by the virtual-lab developer.

InteractiveElements package

The InteractiveElements package includes the ControlElement base class (see Figure 7.5). It is inherited from classes describing interactive control elements; and it inherits from the IViewElement class.

The ControlElement base class includes the code required to:

– Obtain the data sent from the Java program (generated automatically during the model initialization process). The communication will be discussed in Section 7.6.

– Perform the state re-initialization event, which re-initializes the value of the variable defining the state of the element (*var*). This event is triggered when the virtual-lab user manipulates the interactive element.

7.5 Java code generation

There is a relationship among the structure of the Modelica description of the view, the Java code generated and the virtual-lab view obtained by executing this Java code. This relationship is discussed in this section, taking as an example the development of the bouncing-ball virtual-lab (which is included in the *VirtualLabBuilder.Examples* package). The Modelica description of the virtual-lab view and the view obtained by executing the generated Java code are shown in Figures 7.6 and 7.7 respectively.

The Modelica description of the view is built following the methodology described in Section 6.3. It is composed of a set of interactive graphic elements connected following the rules proposed in Section 6.7.

Each interactive graphic element has an "*initial algorithm*" section. This section includes a call to a Modelica function that encapsulates a call to an external C function. This external C function writes in a file the code required to create an instance of the Java class describing the interactive graphic element. The file name is a global parameter (i.e., *inner* to PartialView class and *outer* to the interactive graphic elements).

The PartialView class contains the code required to compile the generated Java application, to pack it in a *jar* file and to execute it. This code is executed during the model initialization process.

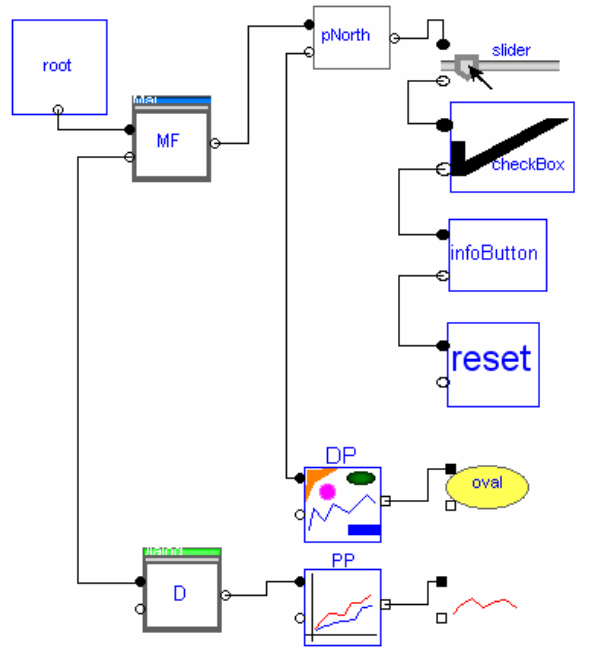


Figure 7.6: Diagram of the view description of the bouncing ball virtual-lab.

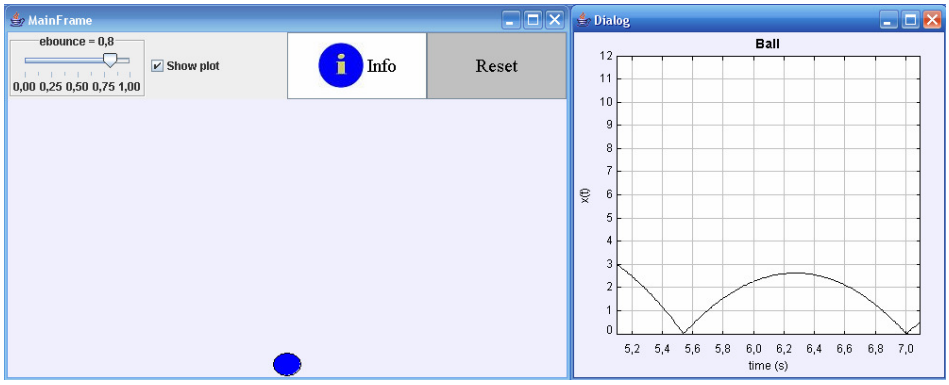


Figure 7.7: Bouncing ball virtual-lab

7.5.1 Execution order of the *initial algorithm* sections

When several interactive graphic elements are used to compose a view, their "*initial algorithm*" sections have to be executed in a sequence that satisfies the rules listed below. The implementation of the interactive elements guarantees that these rules are fulfilled.

1. The "*initial algorithm*" section of the root component is executed in the first place. As a result, the root component writes the first lines of the Java file.

2. The "*initial algorithm*" section of a container is executed before executing the "*initial algorithm*" sections of the components hosted in it.

3. The "*initial algorithm*" section of the drawable components are executed following their drawing order.

4. The "*initial algorithm*" section of the components placed according to certain layout policies within containers are executed in the appropriate order.

The term *path* will be used in the following discussion. This term is used to designate a sequence of interactive graphic elements so that from each interactive graphic element there is a connection to the next element. There are not repeated elements in the path.

For instance, the Modelica description of the view shown in Figure 7.6 contains the following paths:

– *Path 1*: root-MF-pNorth-slider- checkBox-infoButton-reset

– *Path 2*: root-MF-pNorth-DP-oval

– *Path 3*: root-MF-D-PP-trail

The "*initial algorithm*" sections of the interactive graphic elements are executed following a sorted sequence. This sequence is determined by the data dependency among these sections.

The "*initial algorithm*" section of the elements forming a *path*, that has as initial element the root component, are executed in a relative order depending on the distance of the element to the root component. For instance, the order of execution of the "*initial algorithm*" sections in Path 2 is the following: root, MF, pNorth, DP and oval.

The value of the *num* variable of a component indicates the order in which its "initial algorithm" section has been executed. The *num* variable of the root component is equal to zero. The value of the variable *num* of each element of the path satisfies the following relationship (being num_A the number associated to the A component): $num_{root} < num_{MF} < num_{pNorth} < num_{DP} < num_{oval}$.

7.6 Runtime communication between the model simulation and the interactive GUI

The communication established between the C program (generated by Dymola for the Modelica model) and the interactive Java GUI (automatically generated during the initialization process of the virtual-lab described in Modelica) is based on a client-server architecture. The C program is the server and the Java program is the client. The communication is established via TCP sockets.

During the simulation run, there is a bi-directional flow of information between the model simulation and the interactive GUI. At every communication interval:

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

The communication tasks and the classes involved, from the server and the client side, are discussed in this section.

7.6.1 Server side

The following three Modelica partial classes are involved in the communication tasks: PartialView, Drawable and ControlElement. These are the super-classes of the view description in Modelica, the drawable and the interactive elements respectively. The tasks performed by each class are discussed below.

PartialView class

1. To set-up the server. The *startCserver* external C function, included in the *CServer* package, is called to perform this task. This function waits until the client ask for a connection. Then, the connection is established. The function output is the socket description for the established connection.

2. To generate time events at each communication interval (T_{com}), using the built-in $sample(0, T_{com})$ operator. The following two tasks are performed at each time event (see Figure 7.8):

   (a) To call to the *getVarValues* external function, which is included in the *CServer* package. This function receives and processes the data sent by the Java GUI.

   The Java GUI sends the data in a string with the following format:

   $$nChanges, index_1, value_1, ..., index_{nChanges}, value_{nChanges}\#$$

   Where:

   – $nChanges$ is the number of interactive variables modified due to the user's action.

   – $value_i$ is the new value of the *var* variable of the interactive control element number $index_i$.

   The *getVarValues* function receives these data and generates as output the *CK[:]* and $I_{new}[:]$ arrays. These two arrays are global variables (*inner* to PartialView and *outer* to ControlElement class).

   – If $index_i$ is within the string sent by the Java view, then $CK[index_i]$ is set to one. Otherwise, it is equal to zero.

   – If $index_i$ is within the string sent by the Java view, then $I_{new}[index_i]$ is set to $value_i$. Otherwise, $I_{new}[index_i]$ is set to zero.

   (b) To change the value of the boolean variable *refreshView* (from *false* to *true* or vice-versa). This is a global variable (*inner* to PartialView and *outer* to Drawable class).

Drawable class

1. To send information to the Java GUI. Each drawable element sends the following information:

   – The value of the *num* variable of the drawable element.

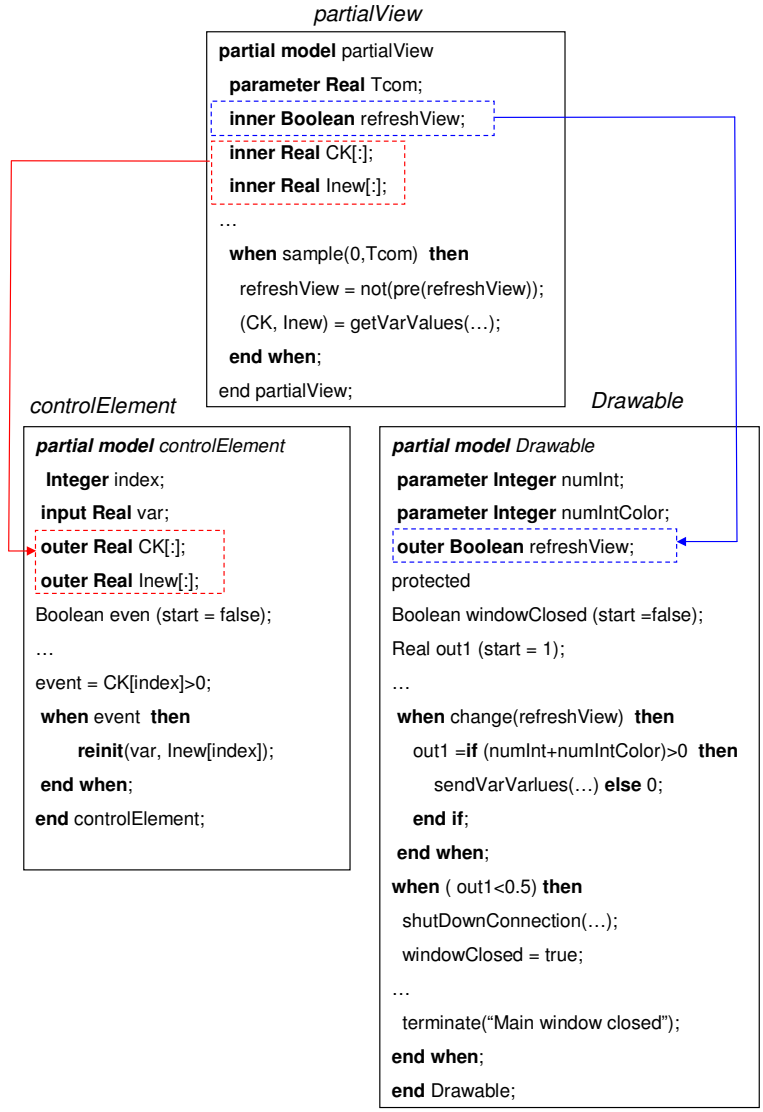


Figure 7.8: Relationship among the PartialView, ControlElement and Drawable classes.

– The geometric properties of the drawable element (i.e., position of the vertices of a polygon, position of the center and length of the axis of an oval, etc.). The value of the *numInt* parameter sets the number of data to be sent. These data are stored in the *vert[:]* vector.

– The color properties of the drawable element (border line and filling color). The value of the *numIntColor* parameter sets the number of data to be sent. These data are stored in the *colors[:]* vector.

This information is sent at each communication interval, when the following two conditions are satisfied: (1) the *refreshView* variable value has changed; and (2) the *windowClosed* variable value is *false*.

2. To end the model simulation when the Java GUI is closed. A *when* clause is triggered when the drawable element sends data to the GUI and, after waiting for T_{Max} seconds, it has not received any reception confirmation from the GUI. This *when* clause performs the following tasks:

   (a) To call to the Modelica built-in operator *terminate*, which finishes the simulation.

   (b) To call to the *shutDownConnection* external function, which is included in the *CServer* package.

   (c) To set *windowClosed* boolean variable to true.

ControlElement class

1. When the value of the *event* variable becomes true, a *when* clause including the code to re-initialize the value of *var* to $I_{new}[index]$ is executed.

   – *var* is the interactive variable.

   – *index* is a number that univocally identifies each interactive control.

   – $CK[:]$ and $I_{new}[:]$ are global variables whose values are transmitted from the PartialView class. $CK[index]$ is equal to one only if the variable associated to the interactive element has been modified due to a user's action. In that case, the new value of the variable is contained in $I_{new}[index]$.

   – *event* is a boolean variable whose value is set to true only when the interactive element has been manipulated by the user ($event = CK[index] > 0$).

7.6.2 Client side

The following two Java classes are involved in the communication tasks: *Client* and *Communication*. They are included in the *graphics.jar* file.

The constructor of the *Client* class contains the code to start the TCP connection with the server. This class includes methods to send and receive data to/from the server.

The *Communication* class includes a *while* loop that is executed until the main window is closed. The following tasks are sequentially executed inside the loop (see Figure 7.9):

1. To refresh the interactive GUI.
2. If the user has manipulated any interactive element, then the following actions are performed:
   (a) The counter of the number of changes ($nChanges$ variable) is increased by one.
   (b) $I_{new}[nChanges]$ is set to the new value.
   (c) $index[nChanges]$ value is set to the identification number of the interactive element that has been manipulated by the user.
3. To call the *sendVarValues* function. This function sends a string with the new values interactively set by the user. The string format was described in Section 7.6.1.
4. If the simulation is paused, then go to step 1, else go to step 5.
5. To obtain the data sent by each drawable element included in the view. For that purpose, the following messages are exchanged with the server:
   – When the client gets ready to receive the data, then it sends a string to the server. The server waits during a limited time (T_{Max}) for the string reception.

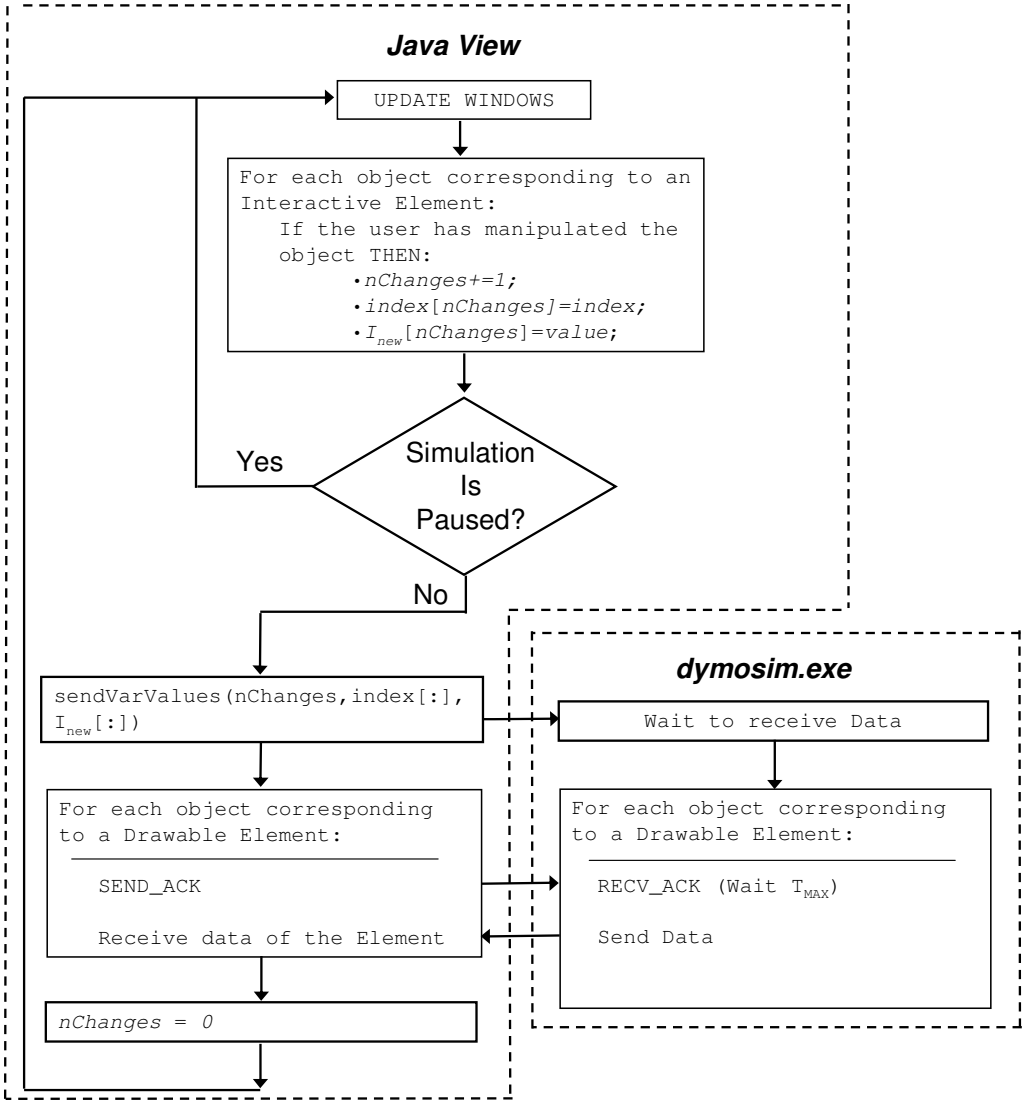


Figure 7.9: Communication between the Java view and the executable file generated by Dymola.

– Once the server has received the string, it sends the value of its *num* variable and the values required to modify the color and the geometric properties of the corresponding Java object. The GUI receives this string and modifies the properties of the corresponding Java object.

6. The value of the *nChanges* variable is set to zero.

7. Go to step 1.

7.7 Conclusions

Chapter 6 was oriented to the *VirtualLabBuilder* users. The information required to build a virtual-lab using the library and some virtual-labs illustrating the proposed approach were discussed.

On the other hand, this chapter was oriented to the library developers. The design and implementation details of the library were discussed. These details are useful in order to create new components and to get a better understanding of the library.

8

Solar House virtual-lab

8.1 Introduction

The use of *VirtualLabBuilder* Modelica library for the implementation of a virtual-lab describing the thermodynamic behavior of a solar house is discussed in this chapter. The solar house model was developed by Markus Weiner as a part of his M.S. thesis (Weiner 1992, Weiner & Cellier 1993) and it was included in the *BondLib* Modelica library by F.E. Cellier. This Modelica model has been adapted for interactive simulation by using the methodology discussed in Chapter 4. The interactive graphic user-to-model interface has been built by using *VirtualLabBuilder*. The virtual-lab obtained is completely written in Modelica language.

8.2 Description of the solar house virtual-lab

The implementation of a virtual-lab intended to illustrate the thermodynamics of an experimental solar house is discussed. This solar house is located near the airport in Tucson, Arizona, and has a passive solar heating system. The house has four rooms: two bedrooms, a living room and a solarium that collects heat during the winter and releases it during the summer. The living room has an

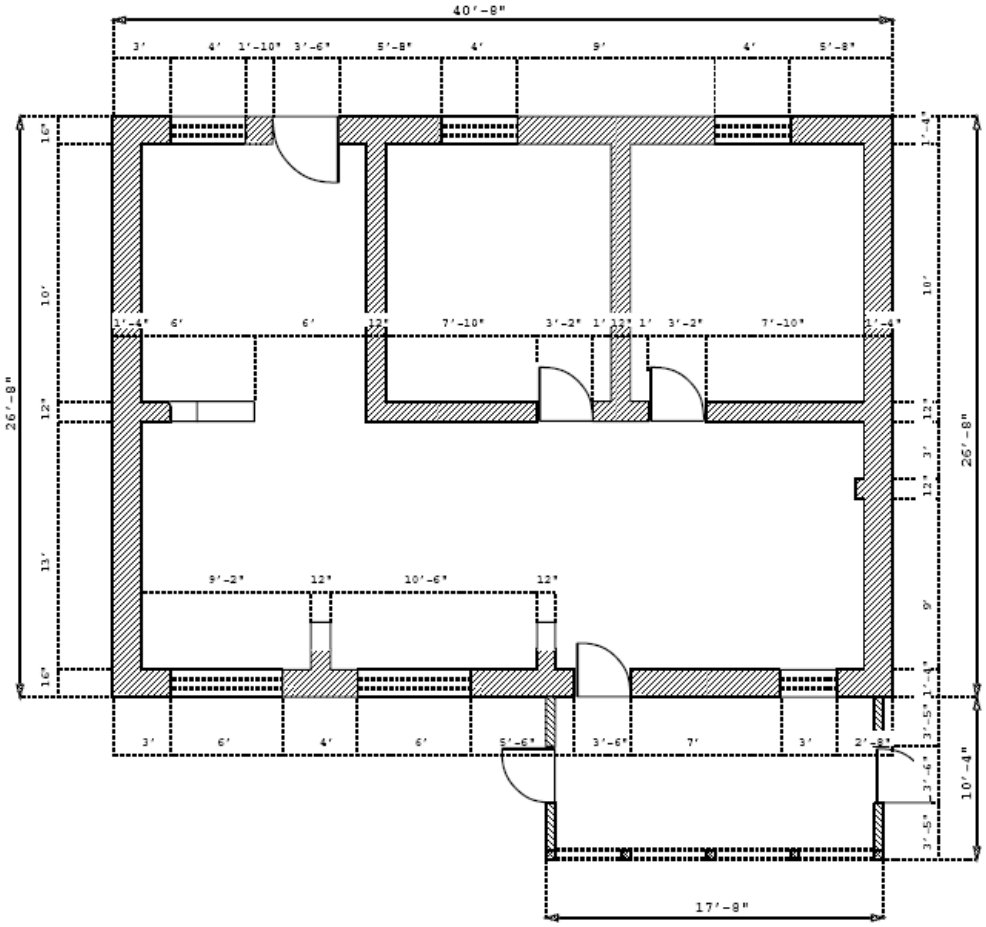


Figure 8.1: Floor plan of the house (Weiner 1992).

air conditioning unit. The floor plan and perspectives of the house are shown in Figures 8.1 and 8.2 respectively (Weiner 1992).

The solar-house virtual-lab allows the user to:

- Change the thermodynamic properties of the slab, the outer and inner walls, and the roof.
- Turn on and off the air conditioning unit, which is placed in the living room.
- Set the parameters of the air conditioning control system (i.e., the setpoints for the minimum and maximum values of the temperature).

The virtual-lab view contains the floor plan of the house (see Figure 8.5b). The room colors change between blue and red as a function of the temperature inside the room. The heat flow through the outer walls are represented by arrows.

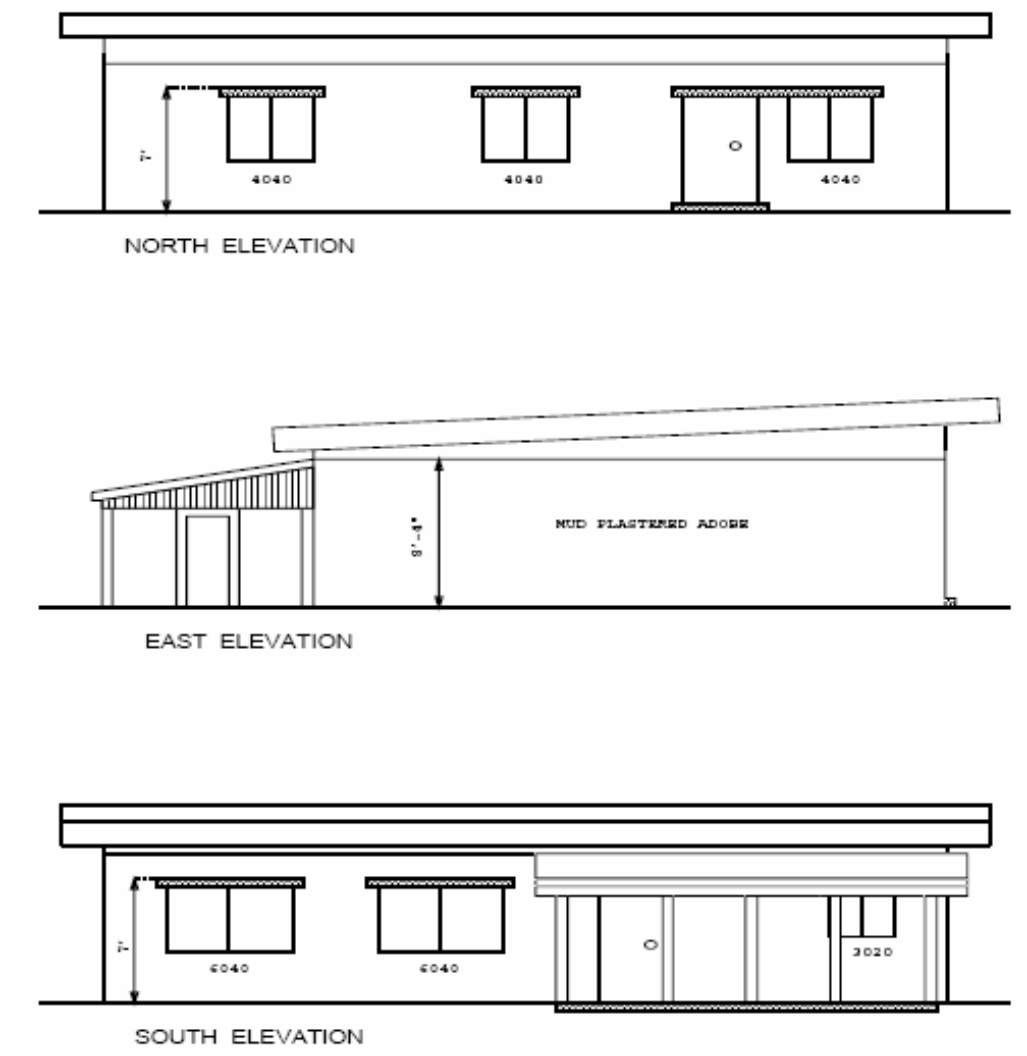


Figure 8.2: Perspectives of the house (Weiner 1992).

The width and orientation of the arrow are functions of the magnitude and the direction of the heat flow, respectively. Also, the virtual-lab view contains plots of some selected variables (see Figure 8.7).

8.3 The Modelica model of the solar house

This solar house model is included within the *Bondlib* library (Cellier & Nebot 2005). The four rooms of the house are composed using models that describe the outer and inner walls, the roofs, the windows, the slabs and the outer and inner doors. A brief description of these models is given below:

Outer wall. This model consists of a boundary convection layer on the outside, three conduction layers inside the wall, and another boundary convection layer on the inside. The model computes its own solar position. The solar radiation model computes the entropy flow to the wall from both direct and diffuse radiation. The ambient air temperature is also computed inside the model.

Inner wall. They have the same structure as the outer walls. However, there is no solar radiation to be taken into account for the interior walls.

Roof. This is exactly the same physical model as the exterior wall model (only with different values for the physical parameters).

Window. This model has an outside convection layer, but no conduction layers, as the glass is considered thin and homogeneous.

Slab. In Tucson, houses are built on sand. The house is not thermally insulated from the ground, thus, the thermal building model ought to take into account the exchange of heat between the house and the slab underneath it. The slab is modeled with a single conduction layer connecting the temperature of the slab to the temperature of the floor. Above the floor, there is a boundary convection layer.

Outer and inner doors. They are similar to windows, in that they are thin and homogeneous. Thus the model contains an outside convection layer, no conduction layers, and no inside convection layer either.

The bond graph technique is used to model the physical laws of heat transfer between the basic components of the house, regarding conduction, convection and radiation. A detailed description of the model can be found in (Weiner 1992, Weiner & Cellier 1993).

8.4 Composing the virtual-lab

The solar house model has been adapted to suit interactive simulation. Interactive parameters and input variables have been re-defined as constant state variables (i.e., with zero time-derivative).

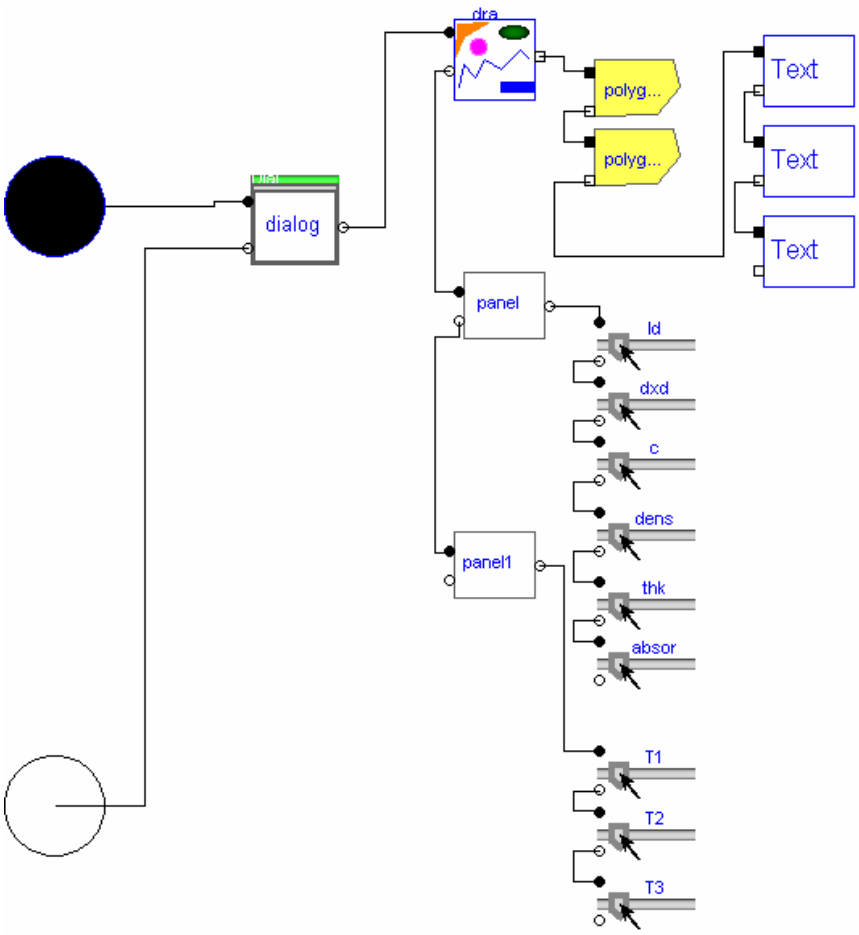
The Modelica description of the virtual-lab view has been developed modularly, by extending and connecting the required graphic components of the *VirtualLabBuilder* library. Modelica classes have been programmed to describe the view associated to an inner wall (InWallView), an outer wall (ExWallView), a slab (SlabView) and a roof (RoofView). These are described next:

- ExWallView class is shown in Figure 8.3a and the graphic interface generated is shown in Figure 8.3b. The ExWallView class contains instances of graphic elements contained in *VirtualLabBuilder* library (i.e., Dialog, DrawingPanel, Panel, Polygon, Text and Slider). The connection among these elements determine the layout of the graphic interface. The graphic interface consists of a window that contains a set of sliders at the bottom and the top (see Figure 8.3b). These sliders allow the user to modify the wall temperature and its thermodynamic properties (i.e., specific thermal conductivity of the dry wall, thickness of the conduction layer, specific heat capacity, density, thickness of the outer wall and absorption coefficient). The center of the window contains a graphical representation of the wall model, which is composed of three conducting layers.

- InWallView class contains sliders that allow the user to change the wall temperature and its thermodynamic properties (i.e., specific thermal conductivity of the dry wall, thickness of the conduction layer, specific heat capacity, density and thickness).

- RoofView class contains sliders that allow the user to change the thermodynamic properties (i.e., specific thermal conductivity, thickness, specific heat capacity and density) of the three conducting layers that compose the roof.

a)



b)

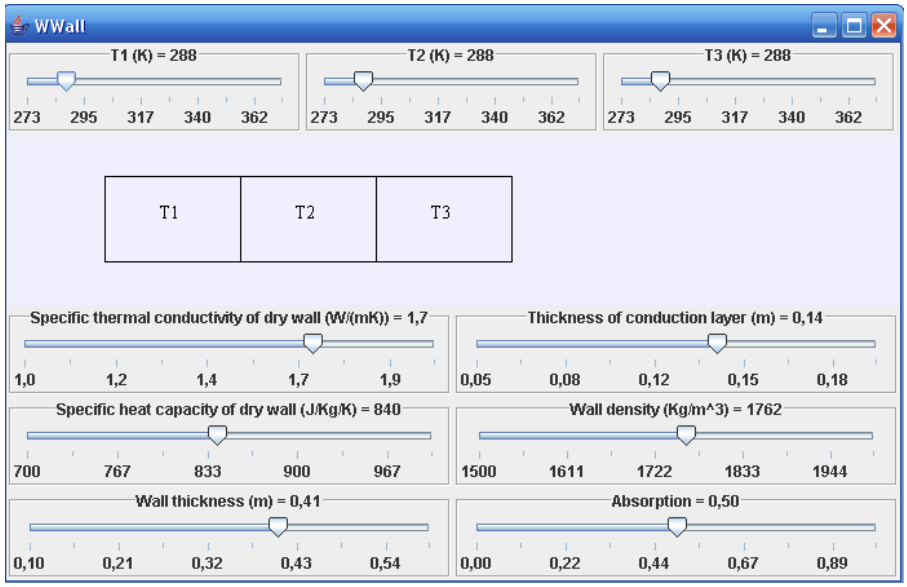


Figure 8.3: ExWallView class: a) diagram of the Modelica description; and b) generated view.

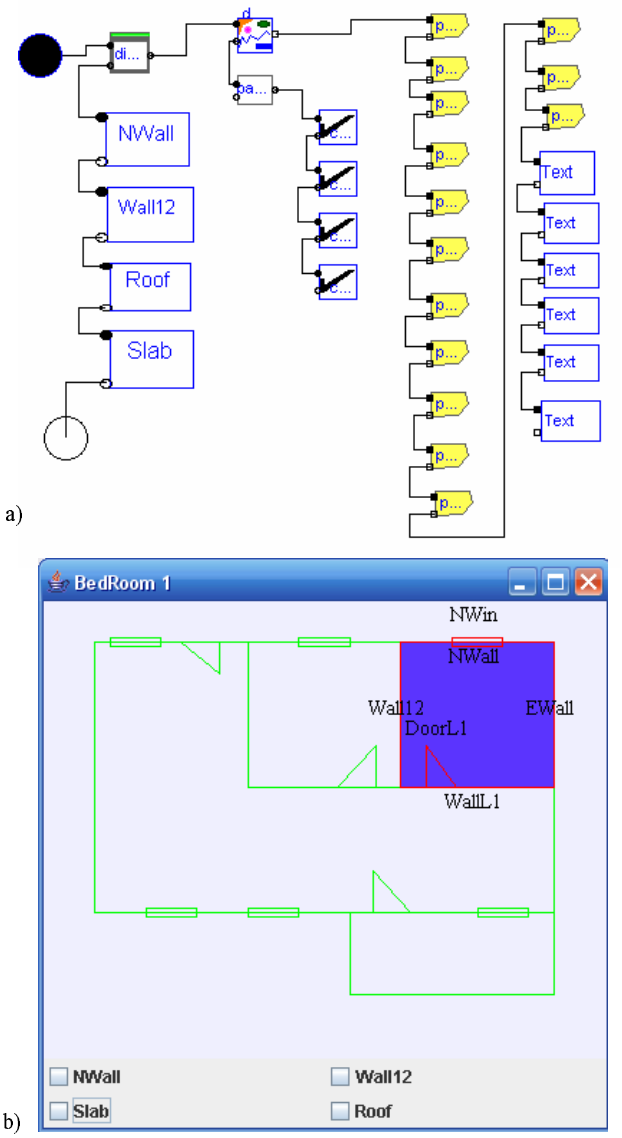


Figure 8.4: BedRoom1View class: a) diagram of the Modelica description; and b) generated view.

- SlabView class contains sliders that allow the user to change its thermodynamic properties (i.e., specific thermal conductivity, thickness of the slab, specific heat capacity, density and thickness of the conduction layer).

Modelica classes have been programmed to describe the view associated to the house (HouseView), the living room (LivingRoomView), and bedrooms 1 and 2 (BedRoom1View and BedRoom2View). These are briefly described next:

- BedRoom1View class is shown in Figure 8.4a and the graphic interface generated is shown in Figure 8.4b. This model contains instances of SlabView,

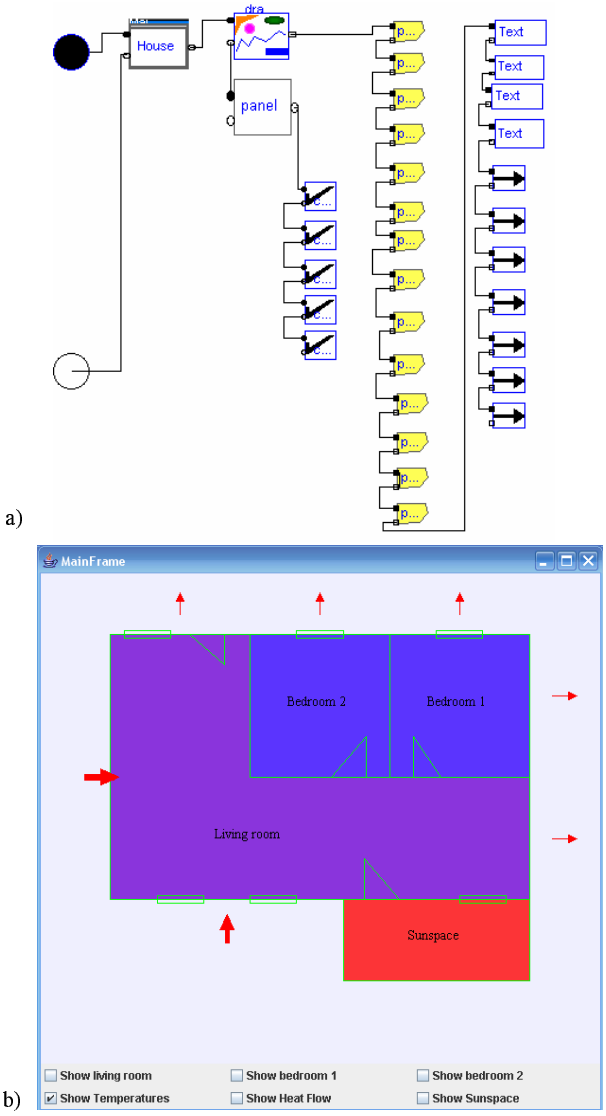


Figure 8.5: HouseView class: a) diagram of the Modelica description; and b) generated view.

RoofView, ExWallView and InWallView classes. The view consists of a window that has a set of checkboxes at the bottom and the floor plan of the room at the center (see Figure 8.4b). The checkboxes allow the user to show and hide the windows associated to each building component of the room (outer and inner walls, slab and roof).

- HouseView class is shown in Figure 8.5a and the graphic interface generated is shown in Figure 8.5b. The view consists of a window that has a set of checkboxes at the bottom and a diagram of the house floor plan in the center (see Figure 8.5b). The checkboxes allow the user to show and hide the

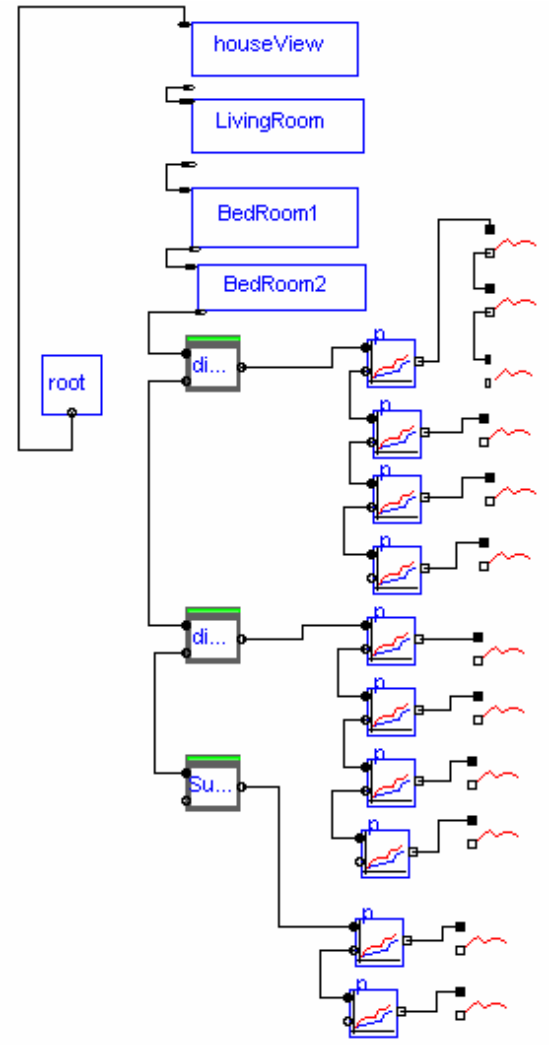


Figure 8.6: Modelica diagram of the complete virtual-lab view.

windows associated to the bedrooms 1 and 2, and to the living room. Each room of the floor plan has a color, that change from blue to red depending on the room temperature. The arrows shown in the floor plan represent the heat flow through the outer walls (see Figure 8.5b). The width and orientation of the arrows depend on the magnitude and the direction of the heat flow, respectively.

The Modelica description of the complete view (i.e, class View) is shown in Figure 8.6. This model extends the PartialView class, which contains: a) one pre-defined graphic element: root; and b) the code required to perform the communication between the model and the view. The View class contains

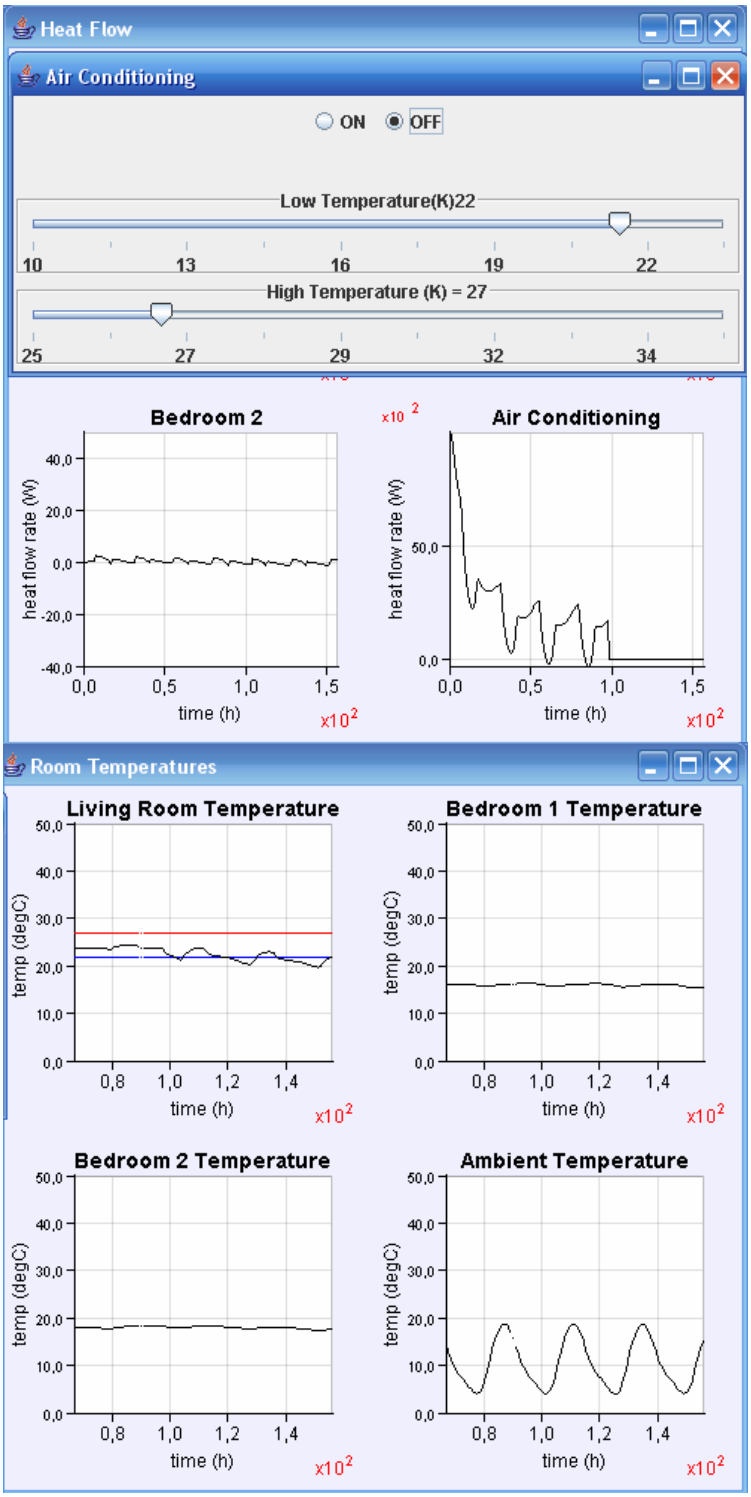


Figure 8.7: Dynamic response of some selected variables.

instances of BedRoom1View, BedRoom2View and LivingRoomView classes. It also contains instances of the *VirtualLabBuilder* library components describing plots. These plots are used to display the time evolution of the heat flow and the temperature in the rooms of the house.

The Modelica description of the virtual-lab has to be an instance of VirtualLab class. This class contains: a) two parametrized generic classes: the classes of the virtual-lab model and view; and b) the equations that link the variables of the model and the view classes.

8.5 Virtual-lab launch

The Modelica description of the virtual-lab is translated using Dymola and run. Then, the jar file containing the Java code of the virtual-lab view is automatically generated and executed. When the jar file is run, 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 dynamic response of the solar house when the air conditioning is turned off is shown in Figure 8.7. This change has been interactively performed by the virtual-lab user at the simulated time 100 h. The following six plots are shown in Figure 8.7:

- The heat flow rate in bedroom 2.
- The heat flow rate of the air conditioning;
- The living room temperature and the setpoint value for the minimum and maximum temperatures.
- The bedroom 1 temperature.
- The bedroom 2 temperature.
- The ambient temperature.

8.6 Conclusions

The feasibility of setting up virtual-labs of complex Modelica models by using *VirtualLabBuilder* has been demonstrated. This approach has two strong points. 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, *VirtualLabBuilder* library allows performing an object-oriented description of the virtual-lab view, which facilitates its development, maintenance and reuse.

VirtualLabBuilder has been used to implement a virtual-lab describing the thermodynamic behavior of a solar house. The model describing the solar house has been adapted to suit interactive simulation. The view has been implemented using graphic elements of *VirtualLabBuilder*.

9

Conclusions and Future Research

9.1 Conclusions

Three different approaches to the implementation of virtual-labs using Modelica language have been proposed:

1. The implementation of virtual-labs with batch interactivity by combining the use of Sysquake and Modelica/Dymola. This work has resulted in the following publications: (Martin et al. 2005b,c).

2. The implementation of virtual-labs with runtime interactivity by combining the use of Ejs and Modelica/Dymola. The obtained results are summarized in the following publications: (Martin et al. 2004a,b, 2005a,b,c).

3. The implementation of virtual-labs with runtime interactivity using only Modelica/Dymola. This approach has been proposed in the following publications: (Martin et al. 2006, Martin-Villalba et al. 2007, Martin et al. 2007).

The methodologies and software tools required to put these three approaches into practice have been developed:

1. A Sysquake to Dymosim interface has been programmed. It consists in a set of functions in LME language which can be called from the Sysquake applications. They are available at `http://www.euclides.dia.uned.es`

2. A methodology for adapting any Modelica model for runtime interactive simulation has been proposed. Two cases have been considered: (1) all interactive quantities can be simultaneously defined as state variables; and (2) several selections of the state variables need to be simultaneously supported.

3. A methodology for combining the use of Ejs and Modelica/Dymola has been proposed. It takes advantage of the existing Ejs-Simulink and Dymola-Simulink interfaces.

4. *VirtualLabBuilder* Modelica library has been designed and programmed. Its on-line documentation is available at `http://www.euclides.dia.uned.es`

The proposed methodology to adapt Modelica models for interactive simulation has been successfully applied to the libraries shown below. Both libraries can be downloaded from `http://www.euclides.dia.uned.es`

1. *JARA* library has been translated into Modelica language and adapted for runtime and batch interactive simulation. This new version of the library is named *JARA 2i*.

2. *tankProcessLAB* Modelica library has been programmed and adapted for runtime and batch interactive simulation.

The proposed approaches have been successfully applied to the development of several virtual-labs for process control education:

1. Virtual-labs with batch interactivity: hysteresis-based controller, chemical reactor, double-pipe heat exchanger and industrial boiler virtual-labs.

2. Virtual-labs with runtime interactivity: quadruple-tank system, industrial boiler, chemical reactor and double-pipe heat exchanger virtual-labs.

Finally, the proposed approach to the implementation of virtual-labs using only Modelica/Dymola has been successfully applied to:

1. The solution of a real industrial problem. A virtual-lab aimed to be applied for testing designs of drum-type washing machines has been implemented. This application has been developed in cooperation with engineers of the Mechanical Engineering Department of the IKERLAN Technological Research Center (Mondragón, Spain).

2. The implementation of a virtual-lab based on a complex Modelica model that has been developed by other authors. A virtual-lab illustrating the thermodynamic behavior of an experimental solar house has been implemented.

9.2 Future research

Finally, some ideas about possible extensions of this work are the following:

- To implement a software tool able to automatically perform the model adaptation for interactive simulation that has been proposed in this dissertation.

- To develop additional interactive graphic elements and to include them in the *VirtualLabBuilder* library. For instance, drawable elements describing 3-D shapes.

- To adapt the libraries included in the Modelica Standard library for interactive simulation and to develop the corresponding graphic interactive elements.

- To explore the use of *VirtualLabBuilder* in other Modelica simulation environments, such as OpenModelica and DrModelica (Lengquist et al. 2003).

- To support the generation of the virtual-labs implemented using *VirtualLabBuilder* as Java applets.

Bibliography

ABACUSS II (2007). ABACUSS II web-site: `http://yoric.mit.edu/abacuss2/abacuss2.html`.

Adams (2007). Adams web-site: `http://www.mscsoftware.com/products/adams.cfm`.

Andersson, M. (1989*a*), An object-oriented modeling environment, *in* 'Proceedings of the 1989 European Simulation Multiconference, The Society for Computer Simulation International', Rome, Italy, pp. 77–82.

Andersson, M. (1989*b*), *Omola - An Object-Oriented Modelling Language*, Report TFRT 7417, Dept of Automatic Control, Lund Institute of Technology, Sweden.

Andersson, M. (1990), *Omola - An Object-Oriented Language for Model Representation*, Licentiate Thesis TFRT 3208, Dept of Automatic Control, Lund Institute of Technology, Sweden.

Andersson, M. (1994), *Object-Oriented Modeling and Simulation of Hybrid Systems*, PhD Thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Åström, K. J., Elmqvist, H. & Mattsson, S. E. (1998), Evolution of continuous-time modeling and simulation, *in* 'Proceedings of the 12^{th} European Simulation Multiconference', Manchester, UK, pp. 9–18.

Åström, K. J. & Hagglund, T. (1995), *PID Controllers: Theory, Design and Tuning*, ISA Press.

Augustin, D., Fineberg, M., Johnson, B., Linebarger, R., Sansom, F. & Strauss, J. (1967), 'The SCi continuous system simulation language (CSSL)', *Simulation* **9**, 281–303.

Barton, P. & Pantelides, C. (1994), 'Modeling of combined discrete/continuous processes', *AIChe Journal* **40**, 966–979.

Bird, R. B., Stewart, W. & Lightfoot, E. N. (1975), *Transport Phenomena*, John Wiley & Sons.

Bodson, M. (2003), Fun control experiments with Matlab and a joystick, *in* 'Proceedings of the 42^{nd} IEEE Conference on Decision and Control', Maui, Hawai, USA, pp. 2508–2513.

Brenan, K. E., Campell, S. L. & Petzold, L. R. (1996), *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM.

Breuneuse, A. P. & Broenink, J. F. (1997), 'Modeling mechatronic systems using the SIDOPS+ language', *Simulation Series* **29**(1), 301–306.

Bunks, C., Chancelier, J. P., Delebecque, F., Gomez, C., Goursat, M., Nikoukhah, R. & Steer, S. (1999), *Engineering and Scientific Computing with Scilab*, Birkhauser.

Bunus, P. & Fritzson, P. (2002), Methods for structural analysis and debugging of Modelica models, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 157–165.

Bush, V. (1931), 'The differential analyzer: a new machine for solving differential equations', *Journal of the Franklin Institute* **212**, 447–488.

Casella, F. & Leva, A. (2003), Modelica open library for power plant simulation: design and experimental validation, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 41–50.

Casella, F. & Leva, A. (2006), 'Modelling of thermo-hydraulic power generation processes using Modelica', *Mathematical and Computer Modelling of Dynamical Systems* **12**(1), 19–33.

Cellier, F. E. (1979), *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D. Dissertation. Diss ETH 6483, Zurich, Switzerland.

Cellier, F. E. (1991), *Continuous System Modeling*, Springer-Verlag.

Cellier, F. E., Elmqvist, H., Otter, M. & Taylor, J. H. (1993), Guidelines for modeling and simulation of hybrid systems, *in* 'Proceedings of the 12^{th} IFAC World Congress', Sydney, Australia, pp. 1219–1225.

Cellier, F. E. & Kofman, E. (2006), *Continuous System Simulation*, Springer-Verlag.

Cellier, F. E. & Nebot, A. (2005), The Modelica bond-graph library, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 57–65.

Chancelier, J. P., Delebecque, F., Gomez, C., M.Goursat, Nikoukhah, R. & Steer, S. (2002), *Introduction a Scilab*, Springer-Verlag.

Clauss, C., Leitner, T., Schneider, A. & Schwarz, P. (2000), Modelling of electrical circuits with Modelica, *in* 'Proceedings of the Modelica Workshop', Lund, Sweden.

Cooper, D. & Dougherty, D. (2000), 'A training simulator for computer aided process control education', *Chemical Engineering Education* **34**, 252–257.

Cooper, D., Dougherty, D. & Rice, R. (2003), 'Building multivariable process control intuition using Control Station', *Chemical Engineering Education* **37**, 100–105.

Cooper, D. & Fina, D. (1999), Training simulators enhance process control education, *in* 'Proceedings of the American Control Conference', San Diego, USA, pp. 997–1001.

Cutlip, M. B. & Shacham, M. (1999), *Problem Solving in Chemical Engineering with Numerical Methods*, Prentice-Hall.

Dahlquist, G. (1959), 'Stability and error bound in the numerical integration of ordinary differential equations', *Transactions No. 130 of the Royal Institute of Technology, Stockholm, Sweden* .

Diaz, J. M., Dormido, S. & Aranda, J. (2005), 'Interactive computer-aided control design using quantitative feedback theory: the problem of vertical movement stabilization on a high-speed ferry', *International Journal of Control* **78**(11), 813–825.

Dimmler, M. & Piguet, Y. (2000), Intuitive design of complex real-time control systems, *in* 'Proceedings of the 11^{th} IEEE International Workshop on Rapid System Prototyping (RSP 2000)', Paris, France, pp. 52–57.

Dormido, S. (2004), 'Control learning: present and future', *Annual Reviews in Control* **28**, 115–136.

Dormido, S. & Esquembre, F. (2003), The quadruple-tank process: an interactive tool for control education, *in* 'Proceedings of the European Control Conference', Cambridge, UK.

Dormido, S., Gordillo, F., Dormido-Canto, S. & Aracil, J. (2002), An interactive tool for introductory nonlinear control systems education, *in* 'Proceedings of the 15^{th} IFAC World Congress', Barcelona, Spain.

Dormido, S., Martin, C., Pastor, R., Sanchez, J. & Esquembre, F. (2004), Magnetic levitation system, *in* 'Proceedings of the American Control Conference', Boston, USA.

Dynasim (2006), *Dymola. User's Manual*, Dynasim AB, Lund, Sweden, `http://www.dynasim.com`.

Eborn, J. (1998), *Modelling and Simulation of Thermal Power Plants*, Technical Report - Licenciate Thesis ISRN LUTFD2/TFRT–3219–SE, Dept. of Automatic Control, Lund Institute of Technology, Sweden.

Eborn, J. (2001), *On Model Libraries for Thermo-Hydraulic Applications*, PhD Dissertation, Dept. of Automatic Control, Lund Institute of Technology, Sweden.

EJS (2007). Ejs web-site: `http://fem.um.es/Ejs`.

Elmqvist, H. (1978), *A Structured Model Language for Large Continuous System*, PhD Dissertation TFRT-1015, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist, H., Bruck, D. & Otter, M. (1996), *Dymola. Usert's Manual. Version 3.0*, Dynasim AB, Lund, Sweden.

Elmqvist, H., Cellier, F. E. & Otter, M. (1993), Object-oriented modeling of hybrid systems, *in* 'Proceedings of the ESS'93, European Simulation Symposium', Delft, The Netherlands.

Elmqvist, H., Cellier, F. E. & Otter, M. (1994), Object-oriented modeling of power-electronic circuits using Dymola, *in* 'Proceedings of the CISS - First Joint Conference of International Simulation Societies', Zurich, Switzerland, pp. 156–161.

Elmqvist, H. & Otter, M. (1994), Methods for tearing systems of equations in object-oriented modeling, *in* 'Proceedings of the ESM'94, European Simulation Multiconference', Barcelona, Spain, pp. 326–332.

Elmqvist, H., Otter, M. & Cellier, F. E. (1995), Inline integration: A new mixed symbolic /numeric approach for solving differential–algebraic equation systems, *in* 'Proceedings of the ESM'95, European Simulation Multiconference', Prague, Czech Republic, pp. 23–34.

Elmqvist, H., Tummescheit, H. & Otter, M. (2003), Object-oriented modeling of thermofluid systems, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linkoping, Sweden, pp. 269–286.

Empresarios Agrupados (2007*a*), *EcosimPro - EL Modelling Language*, EA International.

Empresarios Agrupados (2007*b*), *EcosimPro - Mathemathical Algorithms and Simulation Guide*, EA International.

Empresarios Agrupados (2007*c*), *EcosimPro - User Guide*, EA International.

Engelson, V. (2000), *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing*, PhD Dissertation, Dept. of Computer and Information Science, Linkoping University, Linkoping, Sweden.

EPRI (1984), *Modular Modeling System, Theory Manual, MMS-02 Release*, Electric Power Research Institute, Palo Alto, California, USA.

Erenturk, K. (2005), 'Matlab-based guis for fuzzy logic controller design and applications to pmdc motor and avr control', *Computer Applications in Engineering Education* **13**(1), 10–25.

Esquembre, F. (2004), 'Easy Java Simulations: a software tool to create scientific simulations in Java', *Computer Physics Communications* **156**, 109–204.

Fehlberg, E. (1964), 'New high order Runge-Kutta formulas with step size control for systems of first and second order differential equations', *ZAMM* **44**.

Ferreti, F. D. G. & Schiavo, F. (2006), Modelling and simulation of a washing machine, *in* 'Proceedings of the 50^{th} Int. Congress ANIPLA', Rome, Italy.

Fritzson, P. (2004), *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, IEEE Press - Wiley, John & Sons.

Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H. & Karstrom, A. (2002), The Open Source Modelica project, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 297–306.

Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., Broman, D. & Sandholm, A. (2006), OpenModelica - a free open-source environment for system modeling, simulation, and teaching, *in* 'Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design', Munich, Germany.

Fritzson, P. & Engelson, V. (1998), The Open Source Modelica project, *in* 'Proceedings of the 12^{th} European Conference on Object-Oriented Programming', Brussels, Belgium.

Fritzson, P., Viklund, L. & Fritzson, D. (1995), 'High-level mathematical modeling and programming', *IEEE Software* **12**(4), 77–87.

Froment, G. F. & Bischoff, K. B. (1979), *Chemical Reactor Analysis and Design*, John Wiley & Sons, New York, USA.

Gear, C. W. (1971), 'Simultaneous numerical solution of differential-algebraic equations', *IEEE Transactions on Circuit Theory* **CT-18**, 217–225.

Grace, A. C. W. (1991), Simulab, an integrated environment for simulation and control, *in* 'Proceedings of the 1991 American Control Conference', pp. 1015–1020.

Guzman, J. L., Åström, K. J., Dormido, S., Hagglund, T. & Piguet, Y. (2006), Interactive learning modules for PID control, *in* 'Proceedings of the 7^{th} IFAC Advanced Control Education', Madrid, Spain, pp. 1015–1020.

Guzman, J. L., Berenguel, M. & Dormido, S. (2005), 'Interactive teaching of constrained generalized predictive control', *IEEE Control Systems Magazine* **25**(2), 79–85.

Hairer, E., Lubich, C. & Roche, M. (1989), 'The numerical solution of differential-algebraic systems by Runge-Kutta methods', *Lecture notes in Mathematics* **1409**.

Henrichi, P. (1962), *Discrete Variable Methods in Ordinary Differential Equations*, John Wiley & Sons.

Himmelblau, D. M. & Bischoff, K. B. (1992), *Process Analysis and Simulation*, John Wiley & Sons.

IEEE (1997), *Standard VHDL Analog and Mixed-Signal Extensions*, Technical Report IEEE 1076.1. IEEE.

Incropera, F. P. & DeWitt, D. P. (1996), *Fundamentals of Heat and Mass Transfer*, John Wiley & Sons.

Jackson, A. S. (1960), *Analog Computation*, McGraw-Hill.

Jeandel, A., Boudaud., F. & Larivière, E. (1997), *ALLAN Simulation release 3.1 description*, M.DéGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, France, 1997.

Jimoyiannis, A. & Komis, V. (2001), 'Computer simulations in physics teaching and learning', *Computers & Education* **36**, 183–204.

Johansson, K. H. (2000), 'The quadruple-tank process: a multivariable laboratory process with an adjustable zero', *IEEE Transactions on Control Systems Technology* **8**(3), 456–465.

Johansson, M., Gafvert, M. & Åström, K. J. (1998), 'Interactive tools for education in automatic control', *IEEE Control Systems Magazine* **18**(3), 33–40.

Karayanakis, N. M. (1995), *Advanced System Modelling and Simulation with Block Diagram Languages*, CRC Press, Inc.

Karnopp, D. C., Margolis, D. L. & Rosenberg, R. C. (1990), *System Dynamics: A Unified Approach*, Second Edition. John-Wiley & Sons.

Karnopp, D. C. & Rosenberg, R. C. (1968), *Analysis and Simulation of Multiport Systems - The Bond Graph Approach to Physical System Dynamics*, MIT Press, Cambrige, MA, US.

Kielkowski, R. M. (1998), *Inside SPICE*, McGraw-Hill.

Kloas, M., Friesen, V. & Simons, M. (1995), 'Smile - a simulation environment for energy sytems', *System Analysis Modelling Simulation* **18–19**, 503–509.

Korn, G. A. (1989), *Interactive Dynamic-System Simulation*, McGraw-Hill.

Kostic, M. (2000), Interactive simulation with a LabVIEW virtual instrument, *in* 'Proceedings of the NI Annual Conference', Texas, USA.

LabVIEW (2007). LabVIEW web-site: `http://www.ni.com/labview`.

Lara, J. & Alfonseca, M. (2003), 'Visual interactive simulation for distance education', *SIMULATION: Transactions of the Society for Modeling and Simulation International* **79**(1), 19–34.

Laterburg, U. (2001), *LabVIEW in Physics Education*, `http://www.clab.unibe.ch/labview/whitepaper/LV-PhysicsWPScreen.pdf`.

Lengquist, E., Monemar, S., Fritzson, P. & Bunus, P. (2003), DrModelica - an interactive tutoring environment for modelica, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linköping, Sweden, pp. 125–136.

Longchamp, R. (2006), *Commande numérique de systèmes dynamiques*, PPUR, Lausanne, Switzerland.

Martin, C., Urquia, A. & Dormido, S. (2003), SPICELib - modeling and analysis of electric circuits with Modelica, *in* 'Proceedings of the 3^{rd} International Modelica Conference', Linkoping, Sweden, pp. 161–170.

Martin, C., Urquia, A. & Dormido, S. (2004b), JARA 2i - a Modelica library for interactive simulation of physical-chemical processes, *in* 'Proceedings of the European Simulation and Modelling Conference', Paris, France, pp. 128–132.

Martin, C., Urquia, A. & Dormido, S. (2005a), Object-oriented modeling of virtual laboratories for control education, *in* 'Proceedings of the 16^{th} IFAC World Congress', Prague, Czech Republic, pp. Paper code: Th–A22–TO/2.

Martin, C., Urquia, A. & Dormido, S. (2005b), Modelado orientado a objetos de laboratorios virtuales con aplicación a la enseñanza de control de procesos químicos, *in* 'Proceedings of the 1^{st} Congreso Español de Informática (CEDI-EIWISA)', Granada, Spain, pp. 21–26.

Martin, C., Urquia, A. & Dormido, S. (2005c), Modeling of interactive virtual laboratories with Modelica, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 159–168.

Martin, C., Urquia, A. & Dormido, S. (2005d), A distance learning course on virtual-lab implementation for high school science teachers, *in* 'Proceedings of the 6^{th} International Conference on Virtual University', Bratislava, Slovak Republic, pp. 3–8.

Martin, C., Urquia, A. & Dormido, S. (2006), An approach to virtual-lab implementation using Modelica, *in* 'Proceedings of the 20^{th} Annual European Simulation and Modelling Conference', Toulouse, France, pp. 137–141.

Martin, C., Urquia, A. & Dormido, S. (2007), Virtual-lab of a solar house implemented using VirtualLabBuilder Modelica library, *in* 'Proceedings of the Conference on Systems and Control (CSC'2007)', Marrakech, Morocco, p. paper # 130.

Martin, C., Urquia, A., Sanchez, J., Dormido, S., Esquembre, F., Guzman, J. & Berenguel, M. (2004a), Interactive simulation of object-oriented hybrid models, by combined use of Ejs, Matlab/Simulink and Modelica/Dymola, *in* 'Proceedings of the 18^{th} European Simulation Multiconference', Magdeburg, Germany, pp. 210–215.

Martin-Villalba, C., Urquia, A. & Dormido, S. (2007), Implementation of interactive virtual laboratories for control education using Modelica, *in* 'Proceedings of the European Control Conference 2007', Kos, Greece, pp. 2679–2686.

MathModelica (2007). MathModelica web-site: `http://www.mathcore.com/products/mathmodelica`.

Matlab (2007). Matlab web-site: `http://www.Mathworks.com`.

MATRIX$_X$ (2007). MATRIX$_X$ web-site: `http://www.ni.com/matrixx`.

Mattsson, S. E. (1997), On modeling of heat exchangers in Modelica, *in* 'Proceedings of the European Simulation Symposium, ESS'97', Passau, Germany.

Mattsson, S. E., Olsson, H. & Elmqvist, H. (2000), Dynamic selection of states in Dymola, *in* 'Proceedings of the Modelica Workshop', Lund, Sweden, pp. 61–67.

Mattsson, S. E. & Soderlind, G. (1992), A new technique for solving high-index differential equations using dummy derivatives, *in* 'Proceedings of the IEEE Symposium on Computer-Aided Control System Design', California, USA.

Mazaeda, R., Alves, R., Rueda, A., Merino, A., Acebes, L. F. & Prada, C. (2006), Sugar factory simulator for operators training, *in* 'Proceedings of the 7^{th} IFAC Symposium on Advances in Control Education ACE2006', Madrid, Spain.

MGA Software (1996), *ACSL Graphic Modeller - Version 4.1*, MGA Software.

Mitchell, E. E. L. & Gauthier, J. S. (1976), 'Advanced continuous simulation language (ACSL)', *Simulation* pp. 72–78.

MODE.LA (2007). MODEL.LA web-site: `http://www.mit.edu/afs/athena/org/m/modella/`.

Modelica (2005), *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 2.2*, Modelica Association.

Modelica (2007). Modelica Association web-site: `http://www.modelica.org`.

Muñoz-Gómez, L., Alencastre-Miranda, M. & Rudomín, I. (2003), Defining and executing practice sessions in a robotics virtual laboratory, *in* 'Proceedings of the 4^{th} Mexican International Conference on Computer Science (ENCŠ03)', California, Mexico, pp. 159–165.

Nagel, L. (1975), *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, Memorandum ERL-M520, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, USA.

Nagel, L. & Pederson, D. O. (1973), *Simulation Program with Integrated Circuit Emphasis (SPICE)*, Memorandum ERL-M382, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, USA.

Navaratna, C., Dayawansa, W. P. & Martin, C. F. (2001), Virtual control systems laboratory, *in* 'Proceedings of the 40^{th} IEEE Conference on Decision and Control', Florida, USA, pp. 2839–2843.

OOCSMP (2007). OOCSMP web-site: `http://www.ii.uam.es/~jlara/investigacion/download/OOCSMP.html`.

OpenModelica (2007). OpenModelica project web-site: `http://www.ida.liu.se/~pelab/modelica/OpenModelica.html`.

Open Source Physics (2007). Open Source Physics project web-site: `http://www.opensourcephysics.org`.

OrCAD Inc. (1999), *OrCAD PSpice A/D. Reference Guide & User's Guide*, OrCAD, Inc.

Otter, M., Arzen, K. & Dressler, I. (2005), StateGraph - a Modelica library for hierarchical state machines, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 569–578.

Otter, M., Elmqvist, H. & Mattsson, S. E. (2003), The new Modelica MultiBody library, *in* 'Proceedings of the 3^{rd} Int. Modelica Conference', Linkoping, Sweden, pp. 310–330.

Otter, M. & Olsson, H. (2002), New features in Modelica 2.0, *in* 'Proceedings of the 2^{nd} International Modelica Conference', Oberpfaffenhofen, Germany, pp. 7.1–7.12.

Pantelides, C. C. (1988), 'The consistent initialization of differential-algebraic systems', *SIAM J. SCI. STAT. COMPUT.* **9**(2), 213–231.

Piela, P. C. (1989), *ASCEND: An Object-Oriented Environment for Modeling and Analysis*, PhD Thesis EDRC 02-09-89, Engineering Design Reseach Center, Carnegie Mellon University, Pittsburg, PA, USA.

Piguet, Y., Holmberg, U. & Longchamp, R. (1999), Instantaneous performance visualization for graphical control design methods, *in* 'Proceedings of the 14^{th} IFAC World Congress', Beijing, China.

Piguet, Y. & Longchamp, R. (2006), Interactive applications in a mandatory control course, *in* 'Proceedings of the 7^{th} IFAC Advanced Control Education', Madrid, Spain.

Ragazzini, J. R., Randall, R. H. & Russell, F. A. (1947), 'Analysis of problems in dynamics by electric circuits', *Proc. IRE* **35**(5), 444–452.

Ramirez, W. F. (1989), *Computational Methods for Process Simulation*, Butterworths Publishers, Boston, USA.

Sahlin, P., Brign, A. & Sowell, E. F. (1996), *The Neutral Model Format for Building Simulation, Version 3.02*, Technical Report, Dept. of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden.

Saldamli, L. (2002), *PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations*, Licenciate thesis, Department of Computer and Information Science, Linköping University, Sweden.

Saldamli, L. (2005), A framework for describing and solving PDE models in Modelica, *in* 'Proceedings of the 4^{th} International Modelica Conference', Hamburg, Germany, pp. 113–122.

Saldamli, L. (2006), *PDEModelica - A High Level Language for Modeling with Partial Differential Equations*, PhD thesis, Department of Computer and Information Science, Linköping University, Sweden.

Sanchez, J., Dormido, S. & Esquembre, F. (2005a), 'The learning of control concepts using interactive tools', *Computer Applications in Engineering Education* **13**(1), 84–98.

Sanchez, J., Esquembre, F., Martin, C., Dormido, S., Dormido-Canto, R., Dormido-Canto, S. & Pastor, R. (2005b), 'Easy Java Simulations: An open-source tool to develop interactive virtual laboratories using Matlab/Simulink', *International Journal of Engineering Education* **21**(5), 798–813.

Sanchez, J., Morilla, F., Dormido, S., Aranda, J. & Ruiperez, P. (2002), 'Virtual control lab using Java and Matlab: A qualitative approach', *IEEE Control Systems Magazine* **22**(2), 8–20.

Scilab (2007). Scilab web-site `http://www.scilab.org`.

Selfridge, R. G. (1955), Coding a general purpose digital computer to operate as a differential analyzer, *in* 'Proceedings of the 1955 Western Joint Computer Conference, IRE'.

Shah, S. C., Floyd, M. A. & Lehman, L. L. (1985), 'MATRIX$_X$: Control design and model building cae capability', *Computer-Aided Control Systems Engineering* pp. 181–207.

SIMPACK (2007). SIMPACK web-site `http://www.simpack.com`.

Skogestad, S. & Postlethwaite, I. (1996), *Multivariable Feedback Control*, John Wiley & Sons.

Stephanopoulos, G., Henning, G. & Leone, H. (1990), 'MODEL.LA. a modeling language for process engineering. Part I. The formal framework. Part II. Multi-facetted modeling of processing systems', *Comput. Chem. Engng.* **14**, 813–869.

Sysquake (2004), *Sysquake 3. User's Manual*, Calerga Sarl.

Sysquake (2007). Sysquake web-site `http://www.calerga.com/`.

Thoma, J. U. (1990), *Simulation by Bondgraphs*, Springer-Verlag.

Tummescheit, H. (2002), *Design and Implementation of Object-Oriented Model Libraries using Modelica*, PhD Thesis, Dept. of Automatic Control, Lund Institute of Technology, Sweden.

Ugalde-Loo, C. E. (2005), 2x2 individual channel design MATLAB toolbox, *in* 'Proceedings of the 44^{th} IEEE Conference on Decision and Control and European Control Conference ECC 2005', Seville, Spain.

Urquia, A. (2000), *Modelado Orientado a Objetos y Simulación de Sistemas Híbridos en el Ámbito del Control de Procesos Químicos*, PhD Dissertation, Dept. Informatica y Automatica, Facultad de Ciencias, UNED, Madrid, Spain.

Urquia, A. & Dormido, S. (2003), 'Object-oriented design of reusable model libraries of hybrid dynamic systems', *Mathematical and Computer Modelling of Dynamical Systems* **9**(1), 65–118.

Urquia, A., Martin, C. & Dormido, S. (2005), 'Design of SPICELib: a Modelica library for modeling and analysis of electric circuits', *Mathematical and Computer Modelling of Dynamical System* **11**(1), 43–60.

Weiner, M. (1992), *Bond Graph Model of a Passive Solar Heating System*, Ms Thesis, Dept. of Electr. & Comp. Engr, University of Arizona, USA.

Weiner, M. & Cellier, F. E. (1993), Modeling and simulation of a solar energy system by use of bond graphs, *in* 'Proceedings of the 1^{st} SCS International Conference on Bond Graph Modeling', San Diego, California, USA, pp. 301–306.

Wittenmark, B., Haglund, H. & Johansson, M. (1998), 'Dynamic pictures and interactive learning', *IEEE Control Systems Magazine* **18**(3), 26–32.

Zimmer, D. & Cellier, F. E. (2006), The Modelica multi-bond graph library, *in* 'Proceedings of the 5^{th} International Modelica Conference', Vienna, Austria, pp. 559–568.

A

Sysquake - Dymosim Interface

The *sysquakeDymosimInterface* library contains LME functions to experiment with the *dymosim.exe* file. This file is generated by Dymola from the Modelica model. The *sysquakeDymosimInterface* library can be freely downloaded from *http://www.euclides.dia.uned.es*. A description of each function is provided below.

A.1 setExperiment

PURPOSE

To log to a text file the simulation parameters.

USAGE

```
setExperiment( txtFile, StartTime, StopTime, Increment,
               nInterval, Tolerance, MaxFixedStep, Algorithm )
```

PARAMS

txtFile	Name of the file where the simulation parameters are written. By-default value: *dsin1.txt*.
StartTime	Integration start time (and linearization time).
StopTime	Integration end time.
Increment	Communication step size, provided that Increment value is greater than zero.

nInterval	Number of communication intervals, if greater than zero.
Tolerance	Relative precision of signals for simulation, linearization and trimming.
MaxFixedStep	Maximum step size of fixed step size integrators, provided that MaxFixedStep value is greater than 0.0.
Algorithm	Integer (1...28) for selecting the integration algorithm, as described in (Dynasim 2006).

A.2 getInfo

PURPOSE

To execute 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.

USAGE

```
[p, x0, pN, x0N, inputN, outputN] = getInfo
```

PARAMS

p	Vector that contains the parameter values.
x0	Vector that contains the start values of the state variables.
pN	Set of strings, each string representing the name of a parameter.
x0N	Set of strings, each string representing the name of a state variable.
inputN	Set of strings, each string representing the name of an input.
outputN	Set of strings, each string representing the name of an output.

A.3 setValues

PURPOSE To write to a text file the name and the value of the model parameters and the state variables.

USAGE

```
setValues(txtFile, pN, p, x0N, x0)
```

PARAMS

txtFile	Name of the file where the simulation parameters are written. By-default value: *dsin1.txt*).
pN	Set of strings, representing each string the name of a parameter.
p	Vector that contains the parameter values.
x0N	Set of strings, representing each string the name of a state variable.
x0	Vector that contains the start values of the state variables.

A.4 dymosim

PURPOSE To simulate the Dymola model by executing the following command: *dymosim -d dsin.txt iFile oFile*.

USAGE

```
dymosim()
dymosim(iFile, oFile)
```

PARAMS

iFile	Name of the file that contains the simulation parameters. By-default value: *dsin1.txt*
oFile	Name of the file where the results are saved. Using the command tload the results can be loaded in the Sysquake workspace. By-default value: *dsres.txt*

A.5 linearize

PURPOSE

To obtain the linearized model by executing the following command: *dymosim -l iFile oFile*.

USAGE

```
linearize()
linearize(iFile, oFile)
```

PARAMS

iFile	Name of the file that contains the simulation parameters. By-default value: *dsin1.txt*
oFile	Name of the file where the results are saved. Using the command tload the results can be loaded in the Sysquake workspace. By-default value: *dsres.txt*

A.6 tload

PURPOSE

To read the result file, *oFile*, and to store the signal names and the simulation results into N (text matrix) and s (numeric matrix) respectively.

USAGE

```
[N,s] =  tload(oFile)
```

PARAMS

N	Simulation results. N[i] contains the simulation results of the variable whose name is contained in s[i].
s	Matrix that store the signal names as strings.
oFile	Name of the file where the results are loaded. By-default value: *dsres.txt*

A.7 tloadlin

PURPOSE

To load the linear model generated by dymosim from the *txtfile* file (default file name: *dslin.txt*) into the Sysquake workspace.The linear matrix is described by the following equations:

$$der(x) = A * x + B * u$$

$$y = C * x + D * u$$

USAGE

```
[A,B,C,D,xN,uN,yN] = tloadlin(txtfile)
```

PARAMS

A,B,C,D	Matrices of the linear system.
xN	Set of strings, each string representing the name of a state variable.
uN	Set of strings, each string representing the name of an input variable.
yN	Set of strings, each string representing the name of an output variable.

B

Interactive Models

B.1 Perfect gas

```
model perfectGas
   parameter Boolean nIsState;
   parameter Boolean pIsState;
   parameter Boolean TIsState;
   Real n (unit="mol",
           stateSelect= if nIsState then StateSelect.always else StateSelect.default,
           start=20) "Mol number";
   Real p (unit="N.m-2",
           stateSelect=if pIsState then StateSelect.always else StateSelect.default,
           start=1e5) "Gas pressure";
   Real T (unit="K",
           stateSelect=if TIsState then StateSelect.always else StateSelect.default,
           start=300) "Gas temperature";
   Real V (unit="m3", start=1) "Volume";
   Real Cp (unit="J/(Kg.K)", start=5*R/2) "Heat capacity at constant pressure";
   Real Cv (unit="J/(Kg.K)") "Heat capacity at constant volume";
   Real F (unit="mol.s-1") "Input flow";
   Real Tin (unit="K") "Input temperature";
   Real Q (unit="J.s-1") "Heat flow";
   parameter Real R (unit="J/(mol.K)") = 8.31 "Constant of the perfect gases";
protected
   Real U (unit="J", stateSelect = StateSelect.never) "Internal energy";
   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;
   // Interface
   input Real Iparam[2];
   input Real Ivar[3];
   input Real Istate[3];
   Real CKparam;
   Real CKvar;
   Real CKstate;
   output Real O[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[1]);
      reinit(Cp, Iparam[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[1]);
      reinit(Tin, Ivar[2]);
      reinit(Q, Ivar[3]);
   end when;
   // Output signal
   O = { 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[2]);
      reinit(T, Istate[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[1]);
      reinit(T, Istate[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[1]);
      reinit(p, Istate[2]);
   end when;
end perfectGasSS3;


model perfectGasInteractive
   input Real Iparam[2];
   input Real Ivar[3];
   input Real Istate[3];
   input Real CKparam[3];
   input Real CKvar[3];
   input Real CKstate[3];
   input Real Enabled[3];
   output Real O[8];
   output Real Release[1];
   perfectGasSS1 SS1( CKparam = CKparam[1], CKvar = CKvar[1], CKstate = CKstate[1]);
   perfectGasSS2 SS2( CKparam = CKparam[2], CKvar = CKvar[2], CKstate = CKstate[2]);
   perfectGasSS3 SS3( CKparam = CKparam[3], CKvar = CKvar[3], CKstate = CKstate[3]);
equation
   Iparam = SS1.Iparam;
   Istate = SS1.Istate;
   Ivar = SS1.Ivar;
```

```
    Iparam = SS2.Iparam;
    Istate = SS2.Istate;
    Ivar = SS2.Ivar;
    Iparam = SS3.Iparam;
    Istate = SS3.Istate;
    Ivar = SS3.Ivar;
    Release = 4.0;
    O = if Enabled[1] > 0.5 then SS1.O
        else if Enabled[2] > 0.5 then SS2.O
        else if Enabled[3] > 0.5 then SS3.O
        else zeros(size(O, 1));
end perfectGasInteractive;
```

B.2 Chemical reactor

```
model batchReacLiqAtoPInteractive
    // Physical model
    extends PhysicalModel.batchReacLiqAtoP;
    // Interface
    input Real Iparam[7];
    input Real Ivar[10];
    input Real Istate[4];
    input Real CKparam;
    input Real CKvar;
    input Real CKstate;
    output Real O[21];
    output Real Release;
protected
    Boolean CKparamIs0 (start = true, fixed=true);
    Boolean CKvarIs0 (start = true, fixed=true);
    Boolean CKstateIs0 (start = true, fixed=true);
equation
    // Model release
     Release = 1.0;
    // 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(liq.vessel.vesselVolume,Iparam[1]);
       reinit(liq.liquid.section,Iparam[2]);
       reinit(resistTherm.hTSteam,Iparam[3]);
       reinit(resistTherm.hTWater,Iparam[4]);
       reinit(resistTherm.heatExchArea,Iparam[5]);
       reinit(chRAtoP.kCoef[1],Iparam[6]);
       reinit(chRAtoP.kCoef[2],Iparam[7]);
    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(resistTherm.isHeater,Ivar[1]);
       reinit(fluidTemp.isHeater,Ivar[1]);
       reinit(resistTherm.isChiller,Ivar[2]);
```

```
            reinit(fluidTemp.isChiller,Ivar[2]);
            reinit(fluidTemp.tempHeat,Ivar[3]);
            reinit(fluidTemp.tempCool,Ivar[4]);
            reinit(sourceLiqCtrl.flowVSP, -Ivar[5]);
            reinit(sourceLiqCtrl.tempSP,Ivar[6]);
            reinit(sourceLiqCtrl.fractVSP[1],Ivar[7]);
            reinit(sourceLiqCtrl.fractVSP[2],Ivar[8]);
            reinit(sourceLiqCtrl.fractVSP[3],Ivar[9]);
            reinit(chRAtoP.calcConversion,Ivar[10]);
         end when;
         // 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(liq.liquid.massL[1],Istate[1]);
            reinit(liq.liquid.massL[2],Istate[2]);
            reinit(liq.liquid.massL[3],Istate[3]);
            reinit(liq.liquid.tempL,Istate[4]);
         end when;
         // Output variables
         O = { liq.liquid.massL[1], liq.liquid.massL[2], liq.liquid.massL[3],
               liq.liquid.tempL, liq.liquid.liqHeight, liq.liquid.fluidV,
               fluidTemp.sourceTemp, fluidTemp.consumHeater, fluidTemp.consumChiller,
               fluidTemp.isHeater, fluidTemp.isChiller, -liqSource.inMass.massLF[1],
               -liqSource.inMass.massLF[2], -liqSource.inMass.massLF[3],
               -liqSource.totalMassF, liqSource.tempF, -chRAtoP.inMass.massLF[1],
               -chRAtoP.inMass.massLF[2], -chRAtoP.inMass.massLF[3],
               chRAtoP.conversion, chRAtoP.reactionRate[1] };
      end batchReacLiqAtoPInteractive;
```

C

VirtualLabBuilder - User's Reference

This appendix contains the documentation of some packages of the *VirtualLabBuilder* library as it has been generated by Dymola. Only the packages intended to be directly used by virtual-lab developers have been included (i.e., all packages shown in Figure C.1 except the *src* package). Information about equations and components has been omitted.

Complete on-line information about the *VirtualLabBuilder* library is available at `http://www.euclides.dia.uned.es`

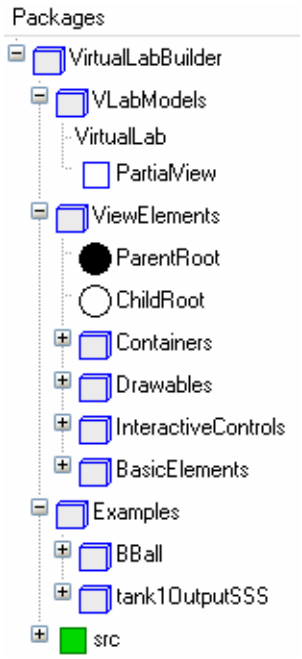


Figure C.1: Packages of *VirtualLabBuilder* library.

VirtualLabBuilder

Information

VirtualLabBuilder- A Modelica library that facilitates the implementation of virtual-labs using only Modelica

Release 1.0 (2007)

Author
Carla Martin-Villalba
Department of Computer Science and Automatic Control, UNED
Madrid, Spain
email: carla@dia.uned.es

VirtualLabBuilder Modelica library facilitates the implementation of virtual-labs using only Modelica. It includes Modelica models implementing graphic interactive elements, such as containers, animated geometric shapes, basic elements and interactive controls. These models allow the virtual-lab developer:

- To compose the view.
- To link the visual properties of the virtual-lab view with the model variables.

The interactive graphic interface is automatically generated during the model initialization process. The components of the library contain the code required to perform the bidirectional communication between the view and the model. In addition, *VirtualLabBuilder* library supports including documentation in the virtual-lab. This documentation is composed of HTML pages.

VirtualLabBuilder Architecture

VirtualLabBuilder library is composed of the packages shown in Figure 1a. Some of them are intended to be used by the virtual-lab developers (i.e., *VirtualLabBuilder* users). These are:

- *ViewElements* and *VLabModels* packages, which contain the classes required to implement the virtual-lab view and to set up the complete virtual-lab.
- *Examples* package, which contains some tutorial material illustrating the library use. The

documentation of these packages is oriented to the *VirtualLabBuilder* users.

On the other hand, the classes within the *src* package are not intended to be directly used by the virtual-lab developers. The documentation of this package describes the implementation details required to modify and extend the *VirtualLabBuilder* library.

In fact, the classes within *ViewElements* and *VLabModels* packages inherit from classes defined within *src* package, inheriting the structure and the behavior, and adding only the documentation oriented to the virtual-lab developer.

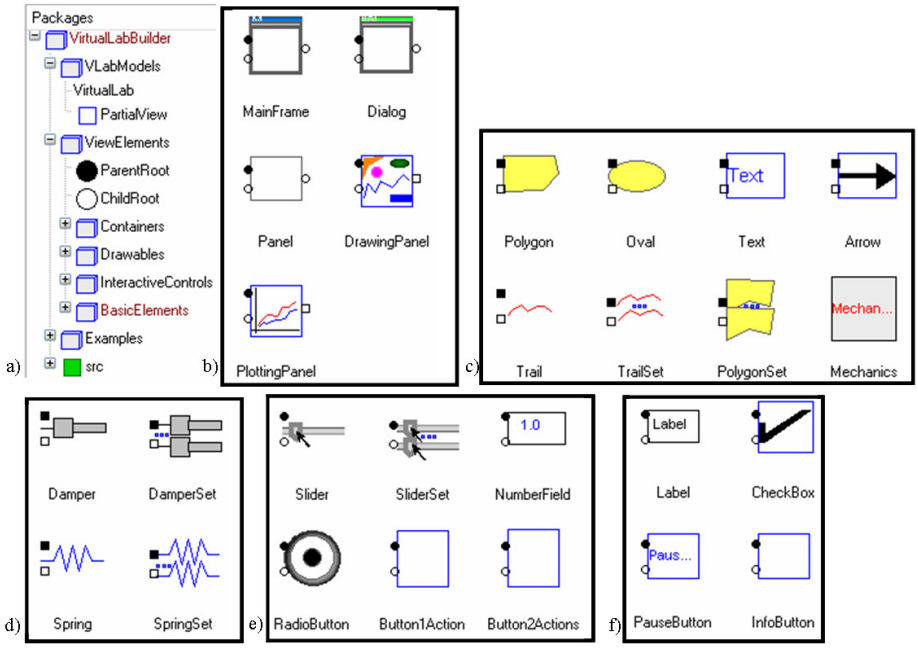


Figure 1.*VirtualLabBuilder* library: a) general structure; and classes within the following packages: b) Containers; c) Drawables; d) Mechanics; e) InteractiveControls; and f) BasicElements..

Steps to describe a virtual-lab

The virtual-lab definition includes the description of the introduction, the model, the view, and the bidirectional flow of information between the model and the view. The virtual-lab definition process is outlined next.

1. **Virtual-lab model.** Any Modelica model can be transformed into other Modelica model suitable for interactive simulation. Essentially, the proposed methodology

consists in modifying the model so that all the variables that need to be changed interactively during the simulation (i.e., the *interactive variables*) are formulated as state variables. In particular, parameters are redefined as time-dependent variables whose time-derivative is equal to zero. Input variables are reformulated analogously in order to become interactive variables. Modelica's *when* clause and *reinit* operator allow describing instantaneous changes in the value of the state variables. This feature is exploited in order to perform the instantaneous changes in the value of the interactive variables produced by the user's interaction. Some of these model manipulations could be performed automatically by a software tool. However, at the present time, they have to be carried out manually by the virtual-lab developer.

2. **Virtual-lab view.** The virtual-lab developer has to define a Modelica class describing the virtual-lab view. This class has to extend another class, named *PartialView*, that is included in *VirtualLabBuilder* library (see Figure 1a). The communication interval (i.e., time interval between to consecutive model-view communications) is a parameter of the *PartialView* class (T_{com}), that can be set by the virtual-lab developer. *PartialView* class contains a pre-defined component: the *root element* for the view description. The classes describing the graphic components are within the *Containers*, *Drawables*, *InteractiveControls* and *BasicElements* packages of *VirtualLabBuilder* library (see Figures 1b, 1c, 1e and 1f respectively). The virtual-lab designer has to compose the virtual-lab view class by instantiating and connecting the required graphic components. The graphic components have to be connected forming a structure, whose root is the *root element*. The connections among the graphic components determines their layout in the virtual-lab view.
3. **Virtual-lab set up.** The virtual-lab developer has to define a Modelica class describing the complete virtual-lab. This class has to contain an instance of the *VirtualLab* class, which is within the *VirtualLabBuilder* library (see Figure 1a). *VirtualLab* class has the following parameters:
   - Model-to-view communication interval (T_{com}).
   - Name of the java file (the content of this file is generated during the model initialization process).
   - Class describing the virtual-lab model.
   - Class describing the virtual-lab view.

   These two classes have been programmed in Steps 1 and 2 respectively. The virtual-lab designer has to set the value of these parameters by writing the name of these two classes. In addition, he has to specify how the variables of the model and the view Modelica classes are linked. This is accomplished by writing the required Modelica equations inside the Modelica class defining the complete virtual-lab.
4. **Virtual-lab translation and execution.** The virtual-lab developer needs to translate using Dymola an instance of the Modelica class defined in Step 3 into an executable file (i.e., *dymosim.exe* file). The virtual-lab is started by executing this file.
5. **Automatic code generation and run.** At the beginning of the simulation run, some calculations are performed in order to solve the model at the initial time. The *initial sections* of the Modelica model describing the virtual-lab are evaluated. In particular, the *initial sections* of the interactive graphic objects composing the virtual-lab view class 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, packed into a jar file and executed. Also, the communication procedure between the model and the view is automatically set up. This communication is based on a client-server architecture: the C-program generated by Dymola (i.e., *dymosim.exe*, see Step 4) is the server and the Java program (which has been automatically generated during the model initialization) is the client. Once the jar file is executed, 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 model sends the data required to refresh the view and the view sends the value of the variables modified due to a user action at the time instant when the communication is performed. The time interval between two consecutive model-view communications was defined in Step 2.

References

Dynasim (2004): *Dymola. User's Manual.* Dynasim AB.Version 5.3a.

Dynasim (2006): *Dymola. User's Manual. Dymola 6 Additions* Dynasim AB.Version 5.3a.

Martin, C. and A. Urquia and S. Dormido (2007): *Implementation of Interactive Virtual Laboratories for Control Education Using Modelica.* Proceedings of European Control Conference 2007, Kos (Greece), pp. 2679-2686.

Martin, C. and A. Urquia and S. Dormido (2007): *Virtual-lab of a Solar House implemented using VirtualLabBuilder Modelica library.* Proceedings of Conference on Systems and Control (CSC'2007), Marrakech (Morocco), paper #130.

Martin, C., Urquia, A., and Dormido, S. (2004): *An Approach to Virtual-Lab Implementation using Modelica.* In: Proceedings of the 2006 European Simulation and Modelling Conference (ESM'2006), Toulouse (France), pp. 137-141.

Package Content

Name	Description
VLabModels	Classes to describe the virtual-lab view and to set-up the virtual-lab
ViewElements	Package including interactive graphic elements
Examples	Some examples of use
src	Source code

HTML-documentation generated by Dymola Mon Aug 27 18:14:34 2007.

VirtualLabBuilder.VLabModels

Classes to describe the virtual-lab view and to set-up the virtual-lab

Information

VLabModels package

VLabModels package includes the *PartialView* and the *VirtualLab* classes. These two classes are required to describe the virtual-lab view and to set-up the virtual-lab, respectively.

Package Content

Name	Description
VirtualLab	Class containing instances of the model and view description
☐ PartialView	Super-class of the model describing the virtual-lab view

---

VirtualLabBuilder.VLabModels.VirtualLab

Class containing instances of the model and view description

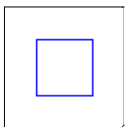
Information

The class describing the complete virtual-lab has to contain an instance of *VirtualLab* class. The virtual-lab designer has to set the name of the model and the view classes.

Parameters

Type	Name	Default	Description
Real	Tcom	0.1	Communication interval
String	fileName	"gui.java"	Name of the java file
replaceable model ViewI		NULL	Class describing the virtual-lab view
replaceable model ModelI		NULL	Class describing the virtual-lab model
String	sourceCodePath	"C:/Program Files/Dymola/Sou...	Path where the C-functions, graphics.jar and delayrun.exe are located

VirtualLabBuilder.VLabModels.PartialView



Super-class of the model describing the virtual-lab view

Information

PartialView class has to be the super-class of the model describing the virtual-lab view. The communication interval (i.e., time interval between to consecutive model-view communications) is a parameter of this class (T_{com}), that can be set by the virtual-lab developer. *PartialView* class contains a pre-defined component: the *root element* for the view description.

Parameters

Type	Name	Default	Description
Real	Tcom	0.1	Communication interval
Integer	serverPort	4242	Server Port number
String	sourceCodePath	"C:/Program Files/Dymola/Sou...	Path where the C-functions, graphics.jar and delayrun.exe are located
String	fileName	"gui.java"	Java file name

HTML-documentation generated by Dymola Mon Aug 27 18:14:34 2007.

VirtualLabBuilder.ViewElements

Package including interactive graphic elements

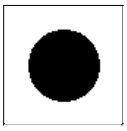
Information

ViewElements package

ViewElements package includes the *Containers*, *Drawables*, *InteractiveControls* and *BasicElements* packages. These packages contain classes describing the interactive graphic elements.

Package Content

Name	Description
ParentRoot	Connector Parent
ChildRoot	Connector Child
Containers	Container elements
Drawables	Drawable elements
InteractiveControls	Interactive control elements
BasicElements	Basic elements



VirtualLabBuilder.ViewElements.ParentRoot

Connector Parent

Information

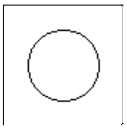
Contents

Type	Name	Description

Integer	nodeReference	Number identifying the component hosting the element
Boolean	borderLayout	True if the component hosting the element has the BorderLayout layout policy

---

VirtualLabBuilder.ViewElements.ChildRoot



Connector Child

Information

Contents

Type	Name	Description
Integer	nodeReference	Number identifying the component
Boolean	borderLayout	True if the component has the BorderLayout layout policy

---

HTML-documentation generated by Dymola Mon Aug 27 20:57:40 2007.

VirtualLabBuilder.ViewElements.Containers

Container elements

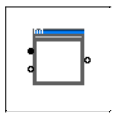
Information

Containers package

Containers package has those graphic elements that are intended to host other graphic elements. The container properties are set in the view definition and they can not be modified during the simulation run.

Package Content

Name	Description
MainFrame	Main window
Dialog	Dialog window
Panel	Panel container
DrawingPanel	Drawing-panel
PlottingPanel	Plotting-panel



VirtualLabBuilder.ViewElements.Containers.MainFrame

Main window

Information

Creates a window where containers, basic elements and interactive controls can be placed. The view can contain only one *MainFrame* object. The user can stop the simulation by closing this window.

Parameters

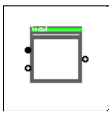
Type	Name	Default	Description
LayoutPolicy	LayoutPolicy	"BorderLayout()"	Layout policy
String	title	"MainFrame"	Text displayed as title

Integer	xPosition	0	X coordinate of the window upper left corner in pixels
Integer	yPosition	0	Y coordinate of the window upper left corner in pixels
Integer	Width	400	Window width in pixels
Integer	Height	400	Window height in pixels
Integer	nRows	1	Number of rows when GridLayout policy is selected
Integer	nColumns	1	Number of columns when GridLayout policy is selected

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components - Parent information
ChildL	cLRight	Connector of non drawable components - Child information
ChildL	cLLeft	Connector of non drawable components - Parent information

---



VirtualLabBuilder.ViewElements.Containers.Dialog

Dialog window

Information

This class, like *MainFrame*, creates a window where containers, basic elements 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.

Parameters

Type	Name	Default	Description
LayoutPolicy	LayoutPolicy	"BorderLayout()"	Layout policy
String	title	"Dialog"	Text displayed as title
Integer	xPosition	0	X coordinate of the window upper left corner in pixels

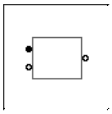
Integer	yPosition	0	Y coordinate of the window upper left corner in pixels
Integer	Width	400	Window width in pixels
Integer	Height	400	Window height in pixels
Integer	nRows	1	Number of rows when GridLayout policy is selected
Integer	nColumns	1	Number of columns when GridLayout policy is selected
String	varName	""	String variable that can be linked to the corresponding variable of a check-box in order to show and hide the window by clicking on the check-box

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components - Parent information
ChildL	cLRight	Connector of non drawable components - Child information
ChildL	cLLeft	Connector of non drawable components - Parent information

---

VirtualLabBuilder.ViewElements.Containers.Panel



Panel container

Information

Panel model creates a panel where containers, basic elements and interactive controls can be placed.

Parameters

Type	Name	Default	Description
LayoutPolicy	LayoutPolicy	"BorderLayout()"	Layout policy
positioninLayout	position	"SOUTH"	If the element hosting the panel has BorderLayout policy, this parameter sets the panel location respect to its container (i.e., north, south, west or east)
Integer	nRows	1	Number of rows if GridLayout policy is selected

Integer	nColumns	1	Number of columns if GridLayout policy is selected

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components - Parent information
ChildL	cLRight	Connector of non drawable components - Child information
ChildL	cLLeft	Connector of non drawable components - Parent information

---

VirtualLabBuilder.ViewElements.Containers.DrawingPane

Drawing-panel

Information

DrawingPanel model creates a two-dimensional container that only can contain drawable 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*, *YMin*) and (*Xmax*, *YMax*)) are parameters of the class whose value can be set by the user.

Parameters

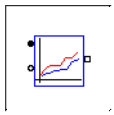
Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container when this container has the BorderLayout layout policy
Real	XMin	-1	Minimum X
Real	XMax	1	Maximum X
Real	YMin	-1	Minimum Y
Real	YMax	1	Maximum Y

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components - Parent information
ChildL	cLLeft	Connector of non drawable components - Parent information

Child	cRight	Connector of drawable components - Child information

---



VirtualLabBuilder.ViewElements.Containers.PlottingPanel

Plotting-panel

Information

PlottingPanel model creates a two-dimensional container with coordinate axes that only can contain drawable objects.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container when this container has the BorderLayout layout policy
String	title	""	Title to display at the top
fontType	font_name	"Dialog"	Title font
Integer	font_size	14	Size of the title font
fontStyle	font_style	"BOLD"	Style of the title font
axesType	axesType	"cartesian2"	The type of axis to be displayed
String	titleX	""	Label of the X axis
xyaxesType	xAxisType	"linear"	The type (linear or log) for cartesian X axis
booleanValue	gridX	"true"	Whether to display the grid for the X axis
String	titleY	""	Label of the Y axis
xyaxesType	yAxisType	"linear"	The type (linear or log) for cartesian Y axis
booleanValue	gridY	"true"	Whether to display the grid for the Y axis
Real	deltaR	2	The separation in R for the polar axis
Real	deltaTheta	3.14159/8	The separation in Theta for polar axis
booleanValue	autoScaleX	"true"	Whether to automatically adjust X scale
Real	marginX	0	Margin to be left in the X scale
booleanValue	autoScaleY	"true"	Whether to automatically adjust Y scale
Real	marginY	0	Margin to be left in the Y scale
Real	minX	0	The minimum X value that can be displayed
Real	maxX	1	The maximum X value that can be displayed

Real	minY	0	The minimum Y value that can be displayed
Real	maxY	1	The maximum Y valued taht can be displayed
booleanValue	coordinates	"true"	Whether to display coordinates when the mouse is pressed
booleanValue	showGrid	"true"	Whether to show or not the grid

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components - Parent information
ChildL	cLLeft	Connector of non drawable components - Parent information
Child	cRight	Connector of drawable components - Child information

---

HTML-documentation generated by Dymola Mon Aug 27 20:57:40 2007.

VirtualLabBuilder.ViewElements.Drawables

Drawable elements

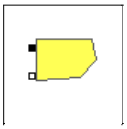
Information

Drawables package

Drawables package contains several classes implementing interactive 2-D shapes, whose properties (i.e., size, position, rotation angle, aspect ratio, colour, etc.) can be linked to the model variables. They are intended to be used for building animated and interactive schematic representations of the system. Objects of *Drawables* classes must be placed inside containers that provide a coordinate system (i.e., containers of *DrawingPanel* and *PlottingPanel* classes).

Package Content

Name	Description
Polygon	Draws a polygon
Oval	Draws an oval
Text	Displays a string
Arrow	Draws an arrow
Trail	Draws a trail
TrailSet	Draws a set of trails
PolygonSet	Draws a set of polygons
Mechanics	Mechanic drawable elements



VirtualLabBuilder.ViewElements.Drawables.Polygon

Draws a polygon

Information

Draws a polygonal curve specified by the coordinates of its vertexes points. The x and y coordinates of the vertexes points of the polygon (*x[:]* and *y[:]* vectors) can be linked to model variables.

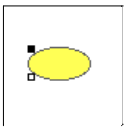
Parameters

Type	Name	Default	Description
booleanValue	filled	"true"	True if the drawable is filled and false otherwise
Color	lineColorp[4]	{0,0,0,255}	The color used for the lines of the component
Color	fillColorp[4]	{0,0,255,255}	The color used to fill the component
Integer	intLineColor	0	1 if the line color changes in time and 0 otherwise
Integer	intFillColor	0	1 if the filling color changes in time and 0 otherwise
Integer	nPoints	1	Number of vertices
booleanValue	closed	"true"	True if the polygon is closed and false otherwise
Integer	intVertexesX[:]	zeros(nPoints)	intVertexesX[i] = 1 if coordinate x of vertex i changes in time
Integer	intVertexesY[:]	zeros(nPoints)	intVertexesY[i] = 1 if coordinate y of vertex i changes in time
Real	stroke	1.0	Stroke used to draw the lines
Integer	gradient	0	1 if there is a gradient in the filling color
Real	p1[2]	{0,0}	Position where the color gradient starts
Color	color1[4]	{192,192,192,255}	Color at point p1
Real	p2[2]	{0,10}	Position where the color gradient finishes
Color	color2[4]	{64,64,64,255}	Color at point p2
booleanValue	cyclic	"true"	True if the color gradient is cyclic

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Oval



Draws an oval

Information

Oval class draws an oval. The position of the oval center (*Center[:]* variable) and the lengths of the axes (*Axes[:]* variable) can be linked to the model variables.

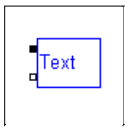
Parameters

Type	Name	Default	Description
booleanValue	filled	"true"	True if the drawable is filled and false otherwise
Color	lineColorp[4]	{0,0,0,255}	The color used for the lines of the component
Color	fillColorp[4]	{0,0,255,255}	The color used to fill the component
Integer	intLineColor	0	1 if the line color changes in time and 0 otherwise
Integer	intFillColor	0	1 if the filling color changes in time and 0 otherwise
Integer	intCenter	0	intCenter = 1 ==> the center changes in time
Integer	intAxes	0	intAxes = 1 ==> the axes change in time
Real	stroke	1.0	Stroke used to draw the lines
Integer	gradient	0	1 if there is a gradient in the filling color
Real	p1[2]	{0,0}	Position where the color gradient starts
Color	color1[4]	{192,192,192,255}	Color at point p1
Real	p2[2]	{0,10}	Position where the color gradient finishes
Color	color2[4]	{64,64,64,255}	Color at point p2
booleanValue	cyclic	"true"	True if the color gradient is cyclic

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Text



Displays a string

Information

Text class displays a string. The position of the string center (*Center[:]* variable) can be linked to the model variables.

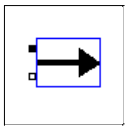
Parameters

Type	Name	Default	Description
Color	textColor[4]	{0,0,0,255}	string color
Integer	intCenter	0	= 0 if the center change in time and 0 otherwise
String	textString	""	String displayed by the element

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Arrow



Draws an arrow

Information

Arrow class displays a vector. The position of the origin (*Origin[:]* variable) and horizontal and vertical components of the vector (*Length[:]* variable) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Color	color[4]	{0,0,0,255}	string color
Integer	intOrigin	0	= 0 if the center change in time and 0 otherwise
Integer	intLength	0	= 0 if the center change in time and 0 otherwise
Real	stroke0	2	Stroke used to draw the lines
Integer	intStroke	0	= 0 if the stroke change in time and 0 otherwise

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Trail

Draws a trail

Information

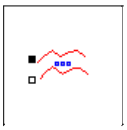
Creates a drawing element that displays a sequence of points at given coordinates of the hosting container. The position of the new point (*point[:]* variable) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Integer	maximumPoints	100	Maximum number of points to be drawn
Integer	nSkip	100	Number of points to skip before plotting one
Color	lineColor[4]	{0,0,0,255}	Line color
booleanValue	connected	"true"	Whether to connect next point with the previous

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components



VirtualLabBuilder.ViewElements.Drawables.TrailSet

Draws a set of trails

Information

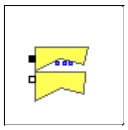
Draws a set of N_trails elements of the *Trail* class. The position of the new point of the trail *i* (*i = 1,..N_trails*)(*point[i,:]* variable) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Integer	N_trails	2	Number of trails
Integer	maximumPoints	100	Maximum number of points to be drawn
Integer	nSkip	1	Number of points to skip before plotting one
Color	lineColor[4]	{0,0,0,255}	Line color
booleanValue	connected	"true"	Whether to connect next point with the previous

Connectors

Type	Name	Description
Parent	parent	Connector of the drawable elements
Child	child	Connector of the drawable elements



VirtualLabBuilder.ViewElements.Drawables.PolygonSet

Draws a set of polygons

Information

Draws a set of N elements of the *Polygon* class. The x and y coordinates of the vertexes points of the polygon *i* (*i = 1,..N*) (*x[i:,:]* and *y[i,:]* vectors) can be linked to model variables.

Parameters

Type	Name	Default	Description
booleanValue	filled	"true"	True if the polygon is filled and false otherwise
Color	lineColorp[4]	{0,0,0,255}	The color used for the lines of the component
Color	fillColorp[4]	{0,0,255,255}	The color used to fill the component
Integer	intLineColor	0	1 if the line color changes in time and 0 otherwise
Integer	intFillColor	0	1 if the filling color changes in time and 0 otherwise
Integer	N	2	Number of polygons
Integer	nPoints	1	Number of vertices
booleanValue	closed	"true"	True if the polygon is closed and false otherwise
Integer	intVertexesX[:]	zeros(nPoints)	intVertexesX[i] = 1 if coordinate x of vertex i changes in time
Integer	intVertexesY[:]	zeros(nPoints)	intVertexesY[i] = 1 if coordinate y of vertex i changes in time
Real	stroke	1	Stroke used to draw the lines
Integer	gradient	0	1 if there is a gradient in the filling color
Real	p1[2]	zeros(2)	Position where the color gradient starts
Color	color1[4]	{192,192,192,255}	Color at point p1
Real	p2[2]	{0,10}	Position where the color gradient finishes
Color	color2[4]	{64,64,64,255}	Color at point p2
booleanValue	cyclic	"true"	True if the color gradient is cyclic

Connectors

Type	Name	Description
Parent	parent	Connector of the drawable elements
Child	child	Connector of the drawable elements

HTML-documentation generated by Dymola Mon Aug 27 18:14:34 2007.

VirtualLabBuilder.ViewElements.Drawables.Mechanics

Mechanic drawable elements

Information

Mechanics package

Mechanics package contains several classes implementing an interactive damper, a set of interactive dampers, an interactive spring and a set of interactive springs.

Package Content

Name	Description
Damper	Draws a damper
DamperSet	Draws a set of dampers
Spring	Draws a spring
SpringSet	Draws a set of springs

VirtualLabBuilder.ViewElements.Drawables.Mechanics.Damper

Draws a damper

Information

Creates a damper. The position of the two damper extremities (*p1[:]* and *p2[:]* variables) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Real	d	1/3	Length of the damper fixed part divided by the damper length
Real	L1	0.02	Distance from the wide to the narrow part of the damper
Real	L2	0.02	Width of the narrow part of the damper
Integer	intX1Y1	0	= 0 if the point (x1, y1) change in time and 0 otherwise
Integer	intX2Y2	0	= 0 if the point (x2, y2) change in time and 0 otherwise
Color	color1[4]	{255,0,255,255}	The damper color changes form this color to color2
Color	color2[4]	{249,204,202,255}	The damper color changes form this color to color1

Connectors

Type	Name	Description

Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Mechanics.DamperSet

Draws a set of dampers

Information

Creates a set of dampers. The position of the two extremities of each damper (*p1[:, :]* and *p2[:, :]* variables) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Integer	N_dampers	2	Number of dampers
Real	d	1/3	length of the damper fixed part divided by the damper length
Real	L1	0.02	distance from the wide to the narrow part of the damper
Real	L2	0.02	width of the narrow part of the damper
Integer	intX1Y1	0	= 0 if the point (x1, y1) change in time and 0 otherwise
Integer	intX2Y2	0	= 0 if the point (x2, y2) change in time and 0 otherwise
Color	color1[4]	{255,0,255,255}	The damper color changes form this color to color2
Color	color2[4]	{249,204,202,255}	The damper color changes form this color to color1

Connectors

Type	Name	Description
Parent	parent	Connector of the drawable elements
Child	child	Connector of the drawable elements

VirtualLabBuilder.ViewElements.Drawables.Mechanics.Spring

Draws a spring

Information

Creates a spring. The position of the two spring extremities (*p1[:]* and *p2[:]* variables) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Real	d	1/19	length of the spring without picks divided by the damper length and two
Integer	N	4	pick number
Real	A	0.05	amplitude of the picks
Integer	intX1Y1	0	= 0 if the point (x1, y1) change in time and 0 otherwise
Integer	intX2Y2	0	= 0 if the point (x2, y2) change in time and 0 otherwise
Real	stroke	2	Stroke used to draw the lines
Color	lineColor[4]	{192,192,192,255}	Line Color

Connectors

Type	Name	Description
Parent	pLeft	Connector of drawable components
Child	cLeft	Connector of drawable components

VirtualLabBuilder.ViewElements.Drawables.Mechanics.SpringSet

Draws a set of springs

Information

Creates a set of dampers. The position of the two extremities of each spring (*p1[:, :]* and *p2[:, :]* variables) can be linked to the model variables.

Parameters

Type	Name	Default	Description
Integer	N_springs	2	Number of springs
Real	d	1/19	length of the spring without picks divided by the damper length and two
Integer	N	4	pick number
Real	A	0.05	amplitude of the picks
Integer	intX1Y1	0	= 0 if the point (x1, y1) change in time and 0 otherwise
Integer	intX2Y2	0	= 0 if the point (x2, y2) change in time and 0 otherwise
Real	stroke	2	Stroke used to draw the lines
Color	lineColor[4]	{192,192,192,255}	Line Color

Connectors

Type	Name	Description
Parent	parent	Connector of the drawable elements
Child	child	Connector of the drawable elements

HTML-documentation generated by Dymola Mon Aug 27 20:57:41 2007.

VirtualLabBuilder.ViewElements.InteractiveControls

Interactive control elements

Information

InteractiveControls package

InteractiveControls package contains classes that allow modifying interactively the value of model variables. Each class includes a definition of an input real variable (*var*) and a boolean variable (*event*). The value of the *event* variable is *true* at those time instants at which the interactive control is manipulated by the virtual-lab user. Otherwise, the *event* variable is *false* The interactive model variable can be linked to the *var* variable by writing the corresponding equation.

Package Content

Name	Description
Slider	Creates a slider
SliderSet	Creates a set of sliders
NumberField	Allows editing a numeric value
RadioButton	Creates a radio-button
Button1Action	Creates a 1 action button
Button2Actions	Creates a 2 actions button

VirtualLabBuilder.ViewElements.InteractiveControls.Slider

Creates a slider

Information

Creates a slider.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
String	stringFormat	"0.00"	Format of the text displayed by the component
String	tickFormat	"0.00"	Format of the text displayed with the ticks
Integer	tickNumber	9	Number of ticks
Real	minimum	0	Minimum value of the variable linked to the component
Real	maximum	1	Maximum value of the variable linked to the component
Real	factor	1	Scale factor
booleanValue	enable	"true"	True if the component is enabled

Connectors

Type	Name	Description

ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

VirtualLabBuilder.ViewElements.InteractiveControls.SliderSet

Creates a set of sliders

Information

Creates a set of sliders.

Parameters

Type	Name	Default	Description
Integer	N	2	Number of sliders
String	stringFormat[N]		Format of the text displayed by the component
String	tickFormat	"0.00"	Format of the text displayed with the ticks
Integer	tickNumber[N]	9*ones(N)	Number of ticks
Real	minimum[N]	zeros(N)	Minimum value of the variable linked to the component
Real	maximum[N]	ones(N)	Maximum value of the variable linked to the component
Real	factor[N]	ones(N)	Scale factor
booleanValue	enable	"true"	True if the component is enabled

Connectors

Type	Name	Description
ParentL	parentL	Connector of drawables
ChildL	childL	Connector of drawables

VirtualLabBuilder.ViewElements.InteractiveControls.NumberField

Allows editing a numeric value

Information

Creates an element that allows editing a numeric value.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
String	stringFormat	"0.00"	Format of the displayed number
booleanValue	enable	"true"	1: enabled, 0: disabled

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

VirtualLabBuilder.ViewElements.InteractiveControls.RadioButton

Creates a radio-button

Information

Creates a radio-button. The *var* variable of this element can have the value 0 or 1.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
booleanValue	buttonValue	"true"	Initial value
String	text	"radioButton"	Text displayed by the element
String	buttonGroup	"buttonGroup"	The radio-button belongs to this group

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

VirtualLabBuilder.ViewElements.InteractiveControls.Button1Action

Creates a 1 action button

Information

Creates a button. The *var* variable is equal to one when the button is pressed and it is equal to zero otherwise. This variable can be used as a condition in a *when* clause. This way, the *when* clause is executed whenever the virtual-lab user presses the button.

Parameters

Type	Name	Default	Description

positioninLayout	position	"CENTER"	Position inside its container
booleanValue	selected	"false"	Whether the button is selected or not
String	label	"info"	Text displayed by the button
alignmentType	alignment	"0.00"	Text alignment
String	image	""	Path of the image of the button
Color	bgcolor[4]	{192,192,192,255}	Background color
String	tooltip	""	Tooltip
Color	lettercolor[4]	{0,0,0,255}	String color
fontType	typeFont	"Times New Roman"	Type of font
fontStyle	styleFont	"Plain"	Style of font
Integer	sizeFont	20	Size of font

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

VirtualLabBuilder.ViewElements.InteractiveControls.Button2Actions

Creates a 2 actions button

Information

This class creates a button. The *var* variable changes alternatively from zero to one and from one to zero whenever the button is pressed. By programming the corresponding *when* clauses, it is possible to associate two different actions to this button: an action is triggered when *var* changes from zero to one, and the other action is triggered when *var* changes from one to zero.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
booleanValue	selected	"false"	Whether the button is selected or not
String	label	"info"	Text displayed by the button
alignmentType	alignment	"0.00"	Text alignment
String	image	""	Path of the image of the button
Color	bgcolor[4]	{192,192,192,255}	Background color
String	tooltip	""	Tooltip
Color	lettercolor[4]	{0,0,0,255}	String color
fontType	typeFont	"Times New Roman"	Type of font
fontStyle	styleFont	"Plain"	Style of font
Integer	sizeFont	20	Size of font

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

HTML-documentation generated by Dymola Mon Aug 27 20:57:41 2007.

VirtualLabBuilder.ViewElements.BasicElements

Basic elements

Information

BasicElements package

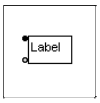
BasicElements package contains classes that can be hosted inside a window or a panel.

Package Content

Name	Description
Label	Decorative label
CheckBox	Check-box
PauseButton	Button to pause and resume the simulation
InfoButton	Button that allows the user to display the virtual-lab documentation

---

VirtualLabBuilder.ViewElements.BasicElements.Label



Decorative label

Information

Creates a decorative label.

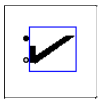
Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
String	text	"text"	Text to be displayed
alignmentType	alignment	"0.00"	Alignment of the text
Color	background[4]	{255,255,255,255}	Background color
Color	foreground[4]	{0,0,0,255}	Foreground color
fontType	typeOfFont	"Times New Roman"	Type of font of the text
fontStyle	styleOfFont	"Plain"	Style of font of the text
Integer	sizeOfFont	10	Size of font

Connectors

Type	Name	Description

ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components



VirtualLabBuilder.ViewElements.BasicElements.CheckBox

Check-box

Information

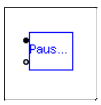
Creates a check-box. The checkbox allows to show or hide the virtual-lab windows by clicking on it.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
booleanValue	initialValue	"false"	Initial value
String	label	""	String displayed by the element
String	varName	"var"	String variable that can be linked to the corresponding variable of a dialog window in order to show and hide the window by clicking on the check-box

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components



VirtualLabBuilder.ViewElements.BasicElements.PauseButton

Button to pause and resume the simulation

Information

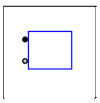
Creates button that allows the user to pause and resume the simulation by clicking on it.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
booleanValue	buttonPause	"false"	Initial value

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components



VirtualLabBuilder.ViewElements.BasicElements.InfoButton

Button that allows the user to display the virtual-lab documentation

Information

Creates a button that allows the user to show or hide a window displaying HTML pages. This feature allows including documentation in the virtual-lab. That is to say, it supports the implementation of the *virtual-lab introduction*.

Parameters

Type	Name	Default	Description
positioninLayout	position	"CENTER"	Position inside its container
booleanValue	selected	"false"	Whether the button is selected or not
String	label	"info"	Text displayed by the button
alignmentType	alignment	"0.00"	Text alignment
String	image	""	Path of the image of the button
Color	bgcolor[4]	{255,255,255,255}	Background color
String	tooltip	""	Tooltip
Color	lettercolor[4]	{0,0,0,255}	String color
fontType	typeFont	"Times New Roman"	Type of font
fontStyle	styleFont	"Plain"	Style of font
Integer	sizeFont	20	Size of font
String	path	""	location of the html file

Integer	xPos	400	Position of the dialog window displayed by the button
Integer	yPos	0	Position of the dialog window displayed by the button
Integer	xWidth	400	Width of the dialog window displayed by the button
Integer	yWidth	400	Height of the dialog window displayed by the button
String	title	"Info window"	Title of the dialog window displayed by the button

Connectors

Type	Name	Description
ParentL	pLLeft	Connector of non drawable components
ChildL	cLLeft	Connector of non drawable components

HTML-documentation generated by Dymola Mon Aug 27 20:57:41 2007.

VirtualLabBuilder.Examples

Some examples of use

Information

Examples package

Example package includes two examples of use.

Package Content

Name	Description
BBall	Bouncing ball
tank	Tank with multiple state selections

HTML-documentation generated by Dymola Tue Aug 28 10:10:29 2007.

VirtualLabBuilder.Examples.BBall

Bouncing ball

Information

BBall package

BBall package includes the description of the bouncing-ball virtual-lab. The *BBModel* has been extracted from (Dynasim 2004) and have been adapted for interactive simulation.

References

Dynasim (2004): *Dymola. User's Manual.* Dynasim AB.Version 5.3a.

Package Content

Name	Description
BBModel	Virtual-lab model
☐ BBView	Virtual-lab view description
BBInteractive	Interactive model

VirtualLabBuilder.Examples.BBall.BBModel

Virtual-lab model

Information

This model has been extracted from (Dynasim 2004). It has been adapted for interactive simulation. The interactive variables of the models are shown in Table 1.

Table 1. Interactive variables.

x	Ball position.
v	Ball velocity.
ebounce	Elasticity coefficient.

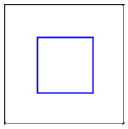
References

Dynasim (2004): *Dymola. User's Manual.* Dynasim AB.Version 5.3a.

Parameters

Type	Name	Default	Description
Height	xStart	10	[m]
Velocity	vStart	0	[m/s]
Mass	m	2	[kg]
Real	ebounceIni	0.8	Initial value of the elasticity coefficient
Real	vsmall	1e-4	

VirtualLabBuilder.Examples.BBall.BBView



Virtual-lab view description

Information

This model has been composed by extending the *PartialView* class and by instantiating and connecting the required components of the *VirtualLabBuilder* library. The connection rules are described in the documentation of the *ViewElements* package.

Parameters

Type	Name	Default	Description
Real	Tcom	0.1	Communication interval
Integer	serverPort	4242	Server Port number
String	sourceCodePath	"C:/Program Files/Dymola/Sou...	Path where the C-functions, graphics.jar and delayrun.exe are located
String	fileName	"gui.java"	Java file name

VirtualLabBuilder.Examples.BBall.BBInteractive

Interactive model

Information

This model has been composed by instantiating the *VirtualLab* Model, which is included in the *VirtualLabBuilder* library. The *BBInteractive* model includes the equations required to link the model and the view variables.

HTML-documentation generated by Dymola Mon Aug 27 20:57:41 2007.

VirtualLabBuilder.Examples.tank

Tank with multiple state selections

Information

tank package

This model intends to illustrate the case of a model supporting several state selections. This model describes a tank with one output at the bottom and one pump placed at the top.

Package Content

Name	Description
tank1OutputModel	Physical model
StateSelection1	h: state variable
StateSelection2	V: state variable
StateSelection3	F: state variable
interactiveModel	Interactive model
□ tank1OutputView	Virtual-lab view
tank1OutputInteractive	Virtual-lab model
setParamVar	When clauses to change interactive parameters and input variables

---

VirtualLabBuilder.Examples.tank.tank1OutputModel

Physical model

Information

This model describes a tank with one output at the bottom and one pump placed at the top. It has been adapted for interactive simulation. The boolean vector *isState[:]*, declared in *tank1OutputModel*, allows controlling the state selection. The size of this vector is equal to the number of interactive time-dependent quantities. The interactive variables of the models are shown in Table 1.

Table 1. Interactive variables.

a	Output hole section.
A	Tank section.
k	Input valve parameter.
vin	Input voltage.

Parameters

Type	Name	Default	Description
Boolean	isState[3]	{true,false,false}	This vector allows controlling the state selection
Boolean	hIsState	isState[1]	true: h is the state variable
Boolean	VIsState	isState[2]	true: V is the state variable
Boolean	FIsState	isState[3]	true: F is the state variable
Real	aInitial	0.4	Initial value of the output hole section
Real	AInitial	2	Initial value of the tank section
Real	KInitial	100	Initial value of the pump parameter
Real	vInitial	0.01	Initial value of the pump voltage
Real	hInitial	3	Initial value of the liquid level
Real	VInitial	20	Initial value of the liquid volume
Real	FInitial	140	Initial value of the liquid flow
Real	g	981	Constant of gravity [cm/s2]

VirtualLabBuilder.Examples.tank.StateSelection1

h: state variable

Information

This model inherits from the *setParamVar* model. The liquid level (*h*) is the state variable of this model. It includes the code to reinitialize the value of the state variable.

Parameters

Type	Name	Default	Description
Boolean	isState[3]	{true,false,false}	This vector allows controlling the state selection

Boolean	hIsState	isState[1]	true: h is the state variable
Boolean	VIsState	isState[2]	true: V is the state variable
Boolean	FIsState	isState[3]	true: F is the state variable
Real	aInitial	0.4	Initial value of the output hole section
Real	AInitial	2	Initial value of the tank section
Real	KInitial	100	Initial value of the pump parameter
Real	vInitial	0.01	Initial value of the pump voltage
Real	hInitial	3	Initial value of the liquid level
Real	VInitial	20	Initial value of the liquid volume
Real	FInitial	140	Initial value of the liquid flow
Real	g	981	Constant of gravity [cm/s2]

VirtualLabBuilder.Examples.tank.StateSelection2

V: state variable

Information

This model inherits from the *setParamVar* model. The liquid volume (*V*) is the state variable of this model. It includes the code to reinitialize the value of the state variable.

Parameters

Type	Name	Default	Description
Boolean	isState[3]	{true,false,false}	This vector allows controlling the state selection
Boolean	hIsState	isState[1]	true: h is the state variable
Boolean	VIsState	isState[2]	true: V is the state variable
Boolean	FIsState	isState[3]	true: F is the state variable
Real	aInitial	0.4	Initial value of the output hole section
Real	AInitial	2	Initial value of the tank section
Real	KInitial	100	Initial value of the pump parameter
Real	vInitial	0.01	Initial value of the pump voltage
Real	hInitial	3	Initial value of the liquid level
Real	VInitial	20	Initial value of the liquid volume
Real	FInitial	140	Initial value of the liquid flow
Real	g	981	Constant of gravity [cm/s2]

VirtualLabBuilder.Examples.tank.StateSelection3

F: state variable

Information

This model inherits from the *setParamVar* model. The liquid flow (*F*) is the state variable of this model. It includes the code to reinitialize the value of the state variable.

Parameters

Type	Name	Default	Description
Boolean	isState[3]	{true,false,false}	This vector allows controlling the state selection
Boolean	hIsState	isState[1]	true: h is the state variable
Boolean	VIsState	isState[2]	true: V is the state variable
Boolean	FIsState	isState[3]	true: F is the state variable
Real	aInitial	0.4	Initial value of the output hole section
Real	AInitial	2	Initial value of the tank section
Real	KInitial	100	Initial value of the pump parameter
Real	vInitial	0.01	Initial value of the pump voltage
Real	hInitial	3	Initial value of the liquid level
Real	VInitial	20	Initial value of the liquid volume
Real	FInitial	140	Initial value of the liquid flow
Real	g	981	Constant of gravity [cm/s2]

VirtualLabBuilder.Examples.tank.interactiveModel

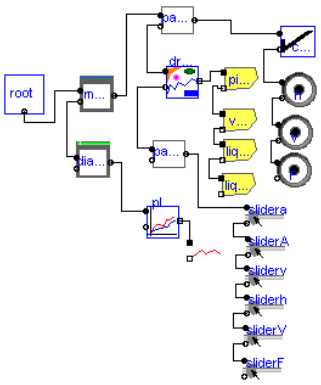
Interactive model

Information

This model is composed of three instantiations of the physical-model: *SS1*, *SS2* and *SS3*. Each model has a different choice of the state variables.

VirtualLabBuilder.Examples.tank.tank1OutputView

Virtual-lab view



Information

This model has been composed by extending the *PartialView* class and by instantiating and connecting the required components of the *VirtualLabBuilder* library. The connection rules are described in the documentation of the *ViewElements* package.

Parameters

Type	Name	Default	Description
Real	Tcom	0.1	Communication interval
Integer	serverPort	4242	Server Port number
String	sourceCodePath	"C:/Program Files/Dymola/Sou...	Path where the C-functions, graphics.jar and delayrun.exe are located
String	fileName	"gui.java"	Java file name
Integer	liquidIX[6]	{1,1,1,1,1,1}	Interactive x components of the liquid polygon
Integer	liquidIY[6]	{1,0,0,1,1,1}	Interactive y components of the liquid polygon

Integer	vaseIX[6]	{1,1,1,1,1,1}	Interactive x components of the vase polygon
Integer	vaseIY[6]	{0,0,0,0,0,0}	Interactive x components of the vase polygon
Integer	liquidFromPipeIx[4]	{1,1,1,1}	Interactive x components of the liquidFromPipe polygon

---

VirtualLabBuilder.Examples.tank.tank1OutputInteractiv

Virtual-lab model

Information

This model has been composed by instantiating the *VirtualLab* Model, which is included in the *VirtualLabBuilder* library. The *tank1OutputInteractive* model includes the equations required to link the model and the view variables.

VirtualLabBuilder.Examples.tank.setParamVar

When clauses to change interactive parameters and input variables

Information

The *setParamVar* class inherits from *physicalModel*, and it contains the *when*-clauses required to change the value of the interactive parameters and input variables. The new values of the interactive quantities, specified interactively by the virtual-lab user, are included in the array *I[:]*. The size of these arrays is equal to the number of interactive parameters plus the number of interactive input variables. The *when*-clauses are triggered by the boolean variables *CK* and *Enabled*. When the value of any of these two variables changes from *false* to *true*, then the interactive quantities are re-initialized to the value of the *I[:]* array.

Parameters

Type	Name	Default	Description
Boolean	isState[3]	{true,false,false}	This vector allows controlling the state selection

Boolean	hIsState	isState[1]	true: h is the state variable
Boolean	VIsState	isState[2]	true: V is the state variable
Boolean	FIsState	isState[3]	true: F is the state variable
Real	aInitial	0.4	Initial value of the output hole section
Real	AInitial	2	Initial value of the tank section
Real	KInitial	100	Initial value of the pump parameter
Real	vInitial	0.01	Initial value of the pump voltage
Real	hInitial	3	Initial value of the liquid level
Real	VInitial	20	Initial value of the liquid volume
Real	FInitial	140	Initial value of the liquid flow
Real	g	981	Constant of gravity [cm/s2]

HTML-documentation generated by Dymola Tue Aug 28 10:19:46 2007.