Ph.D. Dissertation

Adding Interactive Human Interface to Engineering Software

Gonzalo Alberto Farias Castro

Computer Scientist

Departamento de Informática y Automática

Escuela Técnica Superior de Ingeniería Informática

Universidad Nacional de Educación a Distancia

Madrid, 2010

Department Informática y Automática
E.T.S. de Ingeniería Informática

Title Adding Interactive Human Interface
to Engineering Software

Author Gonzalo Alberto Farias Castro

Degree Computer Scientist
Facultad de Ingeniería, Ciencias y Administración
Universidad de la Frontera

Supervisors Sebastián Dormido Bencomo
Francisco Esquembre Martínez

To my family...

Acknowledgements

I wish to thank all those who, in different ways, have contributed to the completion of this thesis:

- To my supervisors Sebastián Dormido Bencomo and Francisco Esquembre Martínez, who have supported me during all these years. I have tried to apply your constructive comments and invaluable guidance to complete this work. Sebastián and Paco, thank you so much for your opinions and suggestions, which have helped me to grow personally and professionally. I can honestly say that I feel extremely lucky to have worked with you.

- To professors Karl-Erik Årzén and Anton Cervin, who have given me the opportunity to work for three months at the Automatic Control department of Lund University. This was a great experience for me, and I tried to do my best to take advantage of your work in order to improve this thesis. Thanks also to all the people in the department who helped me during my stay in the rainy and beautiful green city of Lund. I also would like to thank Karl-Erik to have given me the chance to be a Malmö FF supporter, which was a very gratifying experience.

- To professor Robin De Keyser, who always gave me his support during my time spent at the EeSA Department of Ghent University and also throughout the development of this thesis. Professor, thank you for your successful and positive suggestions. Special thanks also to everyone I met at the UGent: Syafie, Jorge, Luis Alfonso, Clara, Susana, Stefan, Mirabela, Cosmin, Radu, Iounut, and André De Cokere.

- To the people of the DIA Department at the UNED. Thank you very much for your help and support during all these years. Special thanks to Pilar for all the help given with administrative issues. Similarly, I wish to thank professors María Antonia Canto, Joaquín Aranda, Fernando Morilla, José Luis Fernández Marrón, Raquel Dormido, Natividad Duro, Alfonso Urquía, and José Manuel Díaz.

- To my colleagues and friends of the DIA Department at the UNED. Thanks Rocío, Carla, Carlos, Arnoldo, Dictino, Victor, Miguel Angel, David, María, Luis, Alejandro, Jesús, Oscar and Ernesto. Thank you everyone for these really enjoyable

moments that we have shared these years, especially during coffee breaks! Special thanks to my Chilean friends and partners Héctor and Claudia for their constant support and friendship during our stay in Spain. We have been very lucky to coincide here in Spain after finishing university in Chile.

- To professors Sebastián Dormido Canto and José Sánchez Moreno for their friendship and help. Thank you very much for your professional and personal suggestions. Sebas, thank you very much for your insistence in getting a scholarship.

- To the people of the DACYA Department at UCM. Special thanks to professors Matilde Santos and Jesús Manuel de la Cruz, for helping me on my arrival to the UCM. I would like to thank also my dear professor Matilde for her friendship.

- To the people of the CIEMAT, especially Jesús Vega for giving me the opportunity to collaborate and obtain good publications during these years.

- To my friends here in Madrid, Danilo, Desy and Estela, who have always made me feel as if at home in Chile.

- To my dear friend Elizabeth, who started this adventure in Spain with me. Eli, I will always owe you. I hope to have the opportunity to give back at least a small part that you gave me. Eli, people like you make this world a better place.

- To all my family, especially my mother Emelina, my sister Nitcy, my wife Ruth, and my aunts Cecilia and Teresa, for their unconditional support and love. Thank you for never reproaching me all the time that I have not been with you. Tuty this thesis would never have been possible without you.

- To Spain and its people for the financial support obtained for this thesis. Spain, as Neruda said, it will be always in my heart.

- Finally, to God who looks after my family and friends.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past few decades, information and communication technologies (ICTs) are having an increasing impact on education. Widespread technologies such as the Internet, web services, video conferences and others have prompted crucial improvements in education.

The 2001 "Redefining Education" report of the Software & Information Industry Association (SIIA) of the United States of America concludes that *technology is redefining education*. The report states that "the paradigm shift in our education goals and models is just beginning as 21st century solutions merge with 20th century infrastructure and 19th century educational tradition" (SIIA 2001). Such a diagnosis not only acknowledges the tremendous improvements brought by information technologies, but it also reveals the deep changes that the current educational paradigm is facing, moving from a static to a continuous process, better suited for the demand of a life-long learning society (see Figure 1.1).

The use of ICTs provide indeed great opportunities for education. In the last years, the scientific community has made a great effort to take advantage of these new possibilities, developing successful solutions in many different disciplines, including engineering

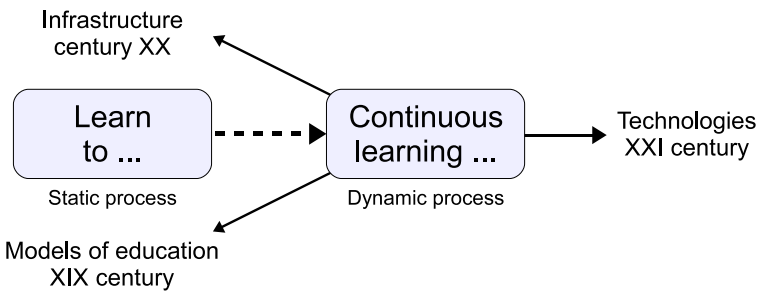


Figure 1.1: Towards a new paradigm of education.

education (Heck 1999, Dormido 2002, Sánchez 2001, Latchman et al. 2001).

Virtual and remote laboratories

In control education the impact of these technologies is even more significant. Experimentation in traditional laboratories is essential for students, who need to understand the fundamental concepts from both perspectives: theoretical and practical. Many examples of traditional laboratories can be found in the literature, see for instance (Leva 2003, 2004, Wellstead 1983, Chandrasekara & Davari 2004, Spong & Block 1995, Radharamanan & Jenkins 2007). However, the high costs associated with equipment, space, and maintenance staff, impose certain constraints on resources, therefore much research has focused on ways to overcome this limitation (Latchman et al. 1999, Gillet et al. 2008, Dormido et al. 2008, Farias et al. 2010, Gomes & Bogosyan 2009, Leva & Donida 2008).

Two of the most important results are *virtual* and *remote laboratories*. Table 1.1 presents a classification of these learning resources based on the following criteria:

The type of resource indicates whether the resource is a real equipment or a model of a physical system.

The type of access indicates whether the resource and the student are in the same or different locations.

Table 1.1: Types of learning resources

		Resource	
		Real	Simulated
Access	Local	Traditional Laboratory	Simulation
	Remote	Remote laboratory	Virtual laboratory

According to these criteria, a traditional laboratory consists of a real resource with local access. Which implies that the student needs access to the location of the real equipment in order to experiment with it.

Simulations, which have become commonplace in the past few years, usually consist of a computer-based model of a real plant. However, in control education, typical

simulations must still be run on the university's computer, because they normally require some specialized, usually expensive, engineering software.

The remote operation of real equipment is commonly referred to as remote laboratory and it allows students to manipulate physical plants, located at the university, from their home computers. This kind of experimentation reduces the time and location constraints of traditional laboratories.

Finally, those simulations that can be run remotely from the student's computer are commonly referred to as virtual laboratories, or distributed simulations.

Both virtual and remote laboratories provide, thanks to remote access, great opportunities for teachers to support the continuous process required for a life-long learning, as mentioned above. This thesis focuses on providing instructors tools to facilitate the creation of virtual and remote laboratories.

Interactivity and visualization

There are two additional aspects of computer simulations relevant to control education: *interactivity* and *visualization* (Dormido, Dormido-Canto, Dormido, Sánchez & Duro 2005, Sánchez et al. 2005). In control engineering simulations, typical analysis of the response of a system comes from the characteristics of its output signals such as the waveform and the period (Uran & Jezernik 2008, Wu et al. 2008). Since the output signals are actually not human-readable, the analysis of the response of the system is neither explicit nor intuitive, therefore without suitable visualization, simulations can be hard-to-understand learning objects. Moreover, much of the analysis is done off-line or statically, which means that signals are obtained and observed only after the simulation has finished, with students rarely interacting with the system, changing parameters or inputs, while the simulation is running. This passive role of students slows down the learning process considerably.

Used properly, interactivity and visualization help minimize these two problems. Instructors can use the graphical power of modern computers to add to simulations a rich level of visual content in order to produce more intuitive and natural learning objects. This sophisticated layer of interactivity and visualization is called human interface.

On-line (or on-the-fly) interaction provides students with the possibility of modifying some inputs or parameters of the system while the simulation is running, which allows

them to better understand the input/output relationships and also to appreciate the influence of specific input on the global response of the system.

Although the importance of interaction is accepted by the engineering education community, its use in actual simulations is not the norm. The main reason is that the development of interactive simulations can be a difficult task from a computer programming point of view. Instructors, who are commonly not programming experts, can encounter problems trying to add user interaction or advanced visualization to existing engineering simulations. This is further complicated by the presence of different computer languages, programming techniques, network protocols, etc.

A second source of frustration is the lack of reusability. Typically, each engineering software has its own framework to add advanced graphical user interfaces (GUIs) to simulations created with it. However, these GUIs are incompatible with other engineering software or even with other simulations generated by the same engineering software. This lack of modularity of human interfaces is common among engineering software.

Both problems are addressed in this thesis, as it introduces a systematic approach to interoperate engineering software and general purpose programming languages. The solution proposed, called **interoperate approach**, allows authors to design and implement separately the GUI and the engineering simulation, connecting them using a standardized protocol. This modularity permits the use of authoring tools that ease the creation of GUIs together with the use of standard engineering software for the simulation of the system. It also promotes the reuse of both simulations and GUIs. The proposed approach is a generalization of previous work described in (Sánchez et al. 2002, 2004, 2005) to connect MATLAB with Java applications. This new approach is used in this thesis to interoperate standard engineering software such as MATLAB, Simulink, Scilab and Sysquake, with interfaces created in Java (Dormido, Esquembre, Farias & Sánchez 2005, Farias et al. 2010, Farias, Keyser, Dormido & Esquembre 2009, Fabregas et al. 2010).

Real-time control systems

In addition to establishing a general approach valid for any engineering simulation, the thesis considers in especial detail the simulation of real-time control systems. These systems are the subject of recent interest since, contrary to the traditional design, a novel

methodology of analysis takes into account the real-time and control aspects together. This new perspective is attractive in itself, but a real-time control system also provides a perfect case study for the interoperate approach thanks to the availability of TrueTime.

TrueTime is a freeware MATLAB/Simulink-based simulator for networked and embedded control systems that has been developed at Lund University since 1999. TrueTime provides mainly two kinds of blocks, TrueTime Kernel and TrueTime Network. These blocks simulate the behaviour of the computer and the network used by the real-time control system. Instructors can then build Simulink simulations by connecting the plant dynamics, modelled using ordinary Simulink blocks, with the input and outputs of the Kernels and Network blocks.

Since TrueTime is a MATLAB toolbox, the interface created in this thesis to interoperate MATLAB and Java was used directly to add interactivity and visualization to the TrueTime simulations. The virtual laboratories of real-time control systems thus developed showed that the interoperate approach works perfectly.

Real-time control systems require a high number of event in order to perform the TrueTime simulation. This has led to the necessity of adding particular methods to the interoperate approach, customized for Simulink models with a high number of events, that can substantially improve performance (Farias, Årzén, Cervin, Dormido & Esquembre 2009, Farias et al. 2007).

The need for a MATLAB license to create and use real-time simulations always brings inconveniences, especially when the simulations must be distributed to students for their study. To minimize this problem, this thesis addressed the creation of a Java library that would reproduce part of the functionality of TrueTime's task model. This Java library, called JavaTrueTime (JTT), does not implement all the functionality that TrueTime provides, but focuses on providing instructors with the basic elements required to build effective real-time control simulations for teaching purposes (Farias, Cervin, Årzén, Dormido & Esquembre 2009, 2008). The thesis shows how to develop plain Java virtual laboratories of real-time control systems using JTT. These examples can be used in any introductory course on real-time control systems.

Experimentation environment

A third topic considered in this thesis is the generation of experiments. The ultimate goal of building a simulation is to execute experiments with it. An experiment consists of extracting data from the model's outputs by modifying its inputs. Typically, engineering simulation tools have a very basic script language to create experiments which does not exploit all the possibilities of modern software technology. Some desirable features, such as adding new events at run-time, or comparing the output plots of two instances of a simulation (in the same graph), are not currently supported by standard simulation tools.

The thesis proposes a basic experimentation environment, a set of tools that allows instructors a wide and very flexible use of their simulations (and others created by colleagues) by manipulating them in the same way that programmers manipulate classes and objects in object-oriented languages. The key features offered by the proposed experimentation environment are discussed in detail in this thesis.

To certify its utility, the proposed experimentation environment has been implemented in the authoring tool Easy Java Simulations. Authors can then use this experimentation environment to build all the functionality required to perform experiments with virtual laboratories (Esquembre et al. 2007). Some examples of use are given in order to demonstrate the power of the resulting implementation.

The three issues discussed in the thesis, **adding interactivity and visualization to engineering simulations**, the **interactive simulation of real-time control systems**, and the **use of an experimentation environment**, are extensively analyzed in order to provide engineering instructors with authoring tools that thoroughly exploit the features of modern technologies.

1.1 Objectives

The general objective of this thesis is to provide authoring tools for instructors to ease the design and building of engineering simulations with pedagogical purposes. Given the extremely wide range of topics to be considered, this thesis focuses on control engineering education. Nevertheless, many of the results obtained in the thesis can be easily applied to other topics in engineering.

The following specific objectives are addressed in this thesis:

- Design an approach to add interactive human interfaces to simulations created with any engineering software.
- Implement libraries to manipulate, locally and remotely, well-known engineering software from Java programs.
- Provide authoring tools to ease the creation of interactive simulations of real-time control systems.
- Define and implement an advanced environment to perform sophisticated experiments with simulations.

These objectives are elaborated on each part of the thesis as described in the following section.

1.2 PhD thesis outline

This work can be divided in three main parts. The first one introduces the key features of an approach to interoperate engineering software with standard programming languages in order to create interactive simulations. A Java implementation of this approach is described in detail. This part ends with examples of complete and sophisticated laboratories created using the libraries developed.

The second part studies the creation of interactive simulations of real-time control systems for teaching purposes. The libraries developed in the previous part are first used to create simulations for the study of these systems using the TrueTime MATLAB toolbox. This part also develops a dedicated library to design Java-based real-time control simulations.

The final part of this thesis focuses on the specification and implementation of an experimentation environment to carry out experiments by flexibly manipulating existing simulations.

Further details of each chapter are given below.

Chapter 2

This chapter describes the key features of the interoperate approach. The idea is to provide a generic way to add interactive human interfaces to simulations created using standard engineering software. The approach helps authors cope with the creation of interactive simulations by dividing the process in two separated stages. The first stage involves the description and implementation of the model using a suitable engineering software. The second stage consists in building the interactive interface using a high-level programming language, which allows final users to interact naturally with the simulation. The link between the user interface and the engineering simulation is accomplished by using the set of methods introduced in this chapter, providing complete control of the engineering simulation from the interactive user interface.

Chapter 3

According to the description of the interoperate approach given in the previous chapter, a set of methods are implemented in Java language to allow manipulation of various well-known engineering software tools. The final result is a group of Java libraries that provides a uniform way to control different engineering software such as MATLAB, Scilab, and Sysquake. A Java program, called **JIM server**, that supports the interoperate approach for remote control of MATLAB/Simulink simulations, is also described in this chapter.

Chapter 4

Once the main libraries for controlling well-known engineering software have been discussed, this chapter focuses on the creation of actual interactive user interfaces. This very important topic is usually a difficult task for instructors, who are not generally experimented programmers. The authoring tool Easy Java Simulations, specifically designed to help building interactive interfaces in Java, is introduced. This tool can make direct use of the libraries developed in the previous chapters in order to manipulate the engineering simulations from an interactive user interface. Although this direct use already helps to build the desired simulations, a deeper integration of the libraries within Easy Java Simulations' architecture was carried out to simplify even more the manipulation of the engineering software.

Chapter 5

The integration of real-time systems and control theory has been treated in recent research by the scientific community. The novel analysis of *co-design* of control and real-time systems requires new tools. One of them is the MATLAB/Simulink-based toolbox called TrueTime (M. Ohlin & Cervin 2007). The tool offers great flexibility and potentiality to perform a deep design and analysis by simulating real-time control systems. However, the simulations obtained using the toolbox lack the interactivity and visualization capabilities required for pedagogical purposes. This chapter uses the results of previous chapters to add interactive human interfaces to TrueTime simulations. The work in this chapter goes a step further and introduces JavaTrueTime, a Java library based on TrueTime's task model, which is specifically designed for simulations with pedagogical purposes. The library reproduces currently only part of the existing functionality of TrueTime, but it does allow authors to build a great variety of interactive simulation of real-time control systems.

Chapter 6

This chapter aims to specify the basic elements of an environment to perform experiments with a simulation. The manipulation of the simulations by using this experimentation environment exploits sophisticated features provided by modern programming languages and technology. This set of elements are implemented in Easy Java Simulations to create a working prototype of the experimentation environment. The chapter includes some simple examples which highlight the main features of the environment. Some aspects of the experiments, particularly interesting from the pedagogical point of view, are also discussed in the chapter.

Chapter 7

The chapter discusses the main conclusions and further work of the research lines described in the thesis.

1.3 Main contributions

The contributions of this thesis are divided in two main aspects: Software components and Publications.

1.3.1 Software components developed

The concrete results of this Ph. D. thesis include the design of protocols and environments, the development of programming libraries, and the creation of a server program and pedagogical laboratories. The most important results are summarised below:

- **Protocols**: Perhaps the most interesting contribution of this thesis consist in the definition of a protocol to implement the interoperate approach. This protocol is based on a number of experiences carried out during years until a final description was established. The protocol allows the interoperation of standard engineering software with general purpose programming languages to make it easy for instructors to create advanced, pedagogically meaningful, reusable interactive simulations of engineering processes. The protocol that supports the experimentation environment described in Chapter 6 is a second contribution of this thesis.

- **Libraries**: Two libraries were developed in this thesis: the Java Internet Matlab Client libray, **JIMC** (Department of Computer Science and Automatic Control, UNED 2010*a*), and the Java TrueTime library, **JTT** (Department of Computer Science and Automatic Control, UNED 2010*c*). The JIMC library can be used by instructors to manipulate the MATLAB/Simulink software from Java programs, according to the specification introduced by the Interoperate Approach described in Chapter 2. The JTT library was developed to ease the creation of interactive simulations for the teaching of real-time control systems. The library is inspired on the MATLAB toolbox TrueTime.

- **Programs**: The most interesting program developed in this thesis is the **JIM server** (Department of Computer Science and Automatic Control, UNED 2010*b*). This software, described in Chapter 3, allows the remote use of the simulations developed with the engineering software MATLAB/Simulink from a Java program.

- **Laboratories**: Many simulations were developed during the doctoral period. Of

special interest are the example simulations used to show the manipulation of MATLAB, Scilab, and Sysquake from Easy Java Simulations. Other interesting simulations are described in Chapter 5, which can be used as part of an introductory course about real-time control systems. Also noteworthy is the remote lab created to perform network-based control of a servo motor. This laboratory has been used in an introductory control course at Ghent University in Belgium.

1.3.2 Publications

During the PhD thesis several articles have been published in specialized journals and international conferences. Many of the papers have been obtained as direct result of this thesis. Others works, however, have been developed in collaborations by the author with different research groups.

Journal papers published

The following publications have been published in journals and are directly related with the PhD thesis:

- G. Farias, K. Årzén, A. Cervin, S. Dormido, F. Esquembre (2010) *Teaching Embedded Control Systems*, International Journal of Engineering Education (accepted for publication).

- G. Farias, R. De Keyser, S. Dormido, F. Esquembre (2009) *Developing Networked Control Labs: A Matlab and Easy Java Simulations Approach*, IEEE Transactions on Industrial Electronics, (accepted, DOI: 10.1109/TIE.2010.2041130).

- N. Duro, R. Dormido, H. Vargas, S. Dormido-Canto, J. Sánchez, G. Farias, F. Esquembre, S. Dormido (2008) *An Integrated Virtual and Remote Control Lab: The Three-Tank System as a Case Study*, Computing in Science and Engineering Magazine, Vol. 10, Issue 4, pp:50-58. Ed: IEEE.

- H. Vargas, J. Sánchez, N. Duro, R. Dormido, S. Dormido-Canto, G. Farias, S. Dormido, F. Esquembre, C. Salzmann, and D. Gillet. (2008) *A Systematic Two-Layer Approach to Develop Web-based Experimentation Environments for Control Engineering Education*, Intelligent Automation and Soft Computing, Vol. 14, Num. 4, pp:505-524, ISSN: 1079-8587.

- R. Dormido, H. Vargas, N. Duro, J. Sánchez, S. Dormido-Canto, G. Farias, F. Esquembre, S. Dormido (2008) *Development of a Web-based Control Laboratory for Automation Technicians: The Three Tank System*, IEEE Transaction on Education, Vol. 51, Num. 1, pp: 35-44.

Journal papers under revision

The following papers are also the result of the thesis, but they are still under the revision process in different international journals.

- G. Farias, A. Cervin, K. Årzén, S. Dormido, F. Esquembre (2009) *Java Simulations of Embedded Control Systems*, Submitted to Real-Time Systems (Springer).
- E. Fabregas, G. Farias, S. Dormido-Canto, S. Dormido, F. Esquembre (2010) *A Practical Approach for Remote Interaction with a Real Plant*, Submitted to Computer & Education (Elsevier).
- J. Sánchez, S. Dormido-Canto, G. Farias, S. Dormido, F. Godoy (2010) *Understanding Automatic Control Concepts by Playing Games*, Submitted to International Journal of Engineering Education.

Papers in conferences

The following articles have been published in national and international conferences mainly related to control engineering. Given the high number of the contributions, only the most important papers are listed.

- G. Farias, R. De Keyser, S. Dormido, F. Esquembre (2009) *Building Remote Labs Using Easy Java Simulation and Matlab*, The European Control Conference 2009, August 23-26, 2009, ISBN: 978-963-311-369-1, Budapest, Hungary.
- G. Farias, A. Cervin, K. Årzén, S. Dormido, F. Esquembre (2008) *Multitasking Real-Time Control Systems in Easy Java Simulations*, Proceedings of the 17th IFAC World Congress 2008, ISBN: 978-1-1234-7890-2, Seoul, Korea.
- G. Farias, S. Dormido, F. Esquembre, H. Vargas, S. Dormido-Canto (2008) *Laboratorio Virtual Para la Enseñanza de Técnicas de Reconocimiento de Patrones*, XIII Latin-American Congress on Automatic Control. Mérida, Venezuela.

- G. Farias, M. Santos, V. López (2008) *Brain Tumour Diagnosis with Wavelets and Support Vector Machines*, 3rd International Conference on Intelligent System and Knowledge Engineering, Proceedings of the 2008 3rd ISKE, IEEE Press, ISBN: 978-1-4244-2197-8, pp: 1453-1459, November 17-19, Xiamen, China.

- F. Esquembre, S. Dormido, G. Farias (2007) *Defining and Performing Experiments in Virtual Laboratories*. International Conference on Engineering Education, ICEE2007. September 3-7. Coimbra Portugal.

- G. Farias, K. Årzén, A. Cervin (2007) *Interactive Real-Time Control Labs with TrueTime and Easy Java Simulations*, In Proceedings of the International Multiconference on Computer Science and Information Technology, International Workshop on Real Time Software, pp.811-820. Wisla, Poland.

- G. Farias, F. Esquembre, J. Sánchez, S. Dormido, H. Vargas, S. Dormido-Canto, R. Dormido, N. Duro (2006) *Laboratorios Virtuales Remotos Usando Easy Java Simulations y Simulink*, Jornadas de Automática. Almería, España.

- H. Vargas, R. Dormido, N. Duro, J. Sánchez, S. Dormido-Canto, G. Farias, S. Dormido, F. Esquembre (2006) *Heatflow: Un laboratorio basado en web usando Easy Java Simulations y Labview para el entrenamiento de técnicas de automatización*, XII Latin-American Congress on Automatic Control. Bahía, Brasil.

- G. Farias, F. Esquembre, J. Sánchez, S. Dormido, H. Vargas, S. Dormido-Canto, R. Dormido, N. Duro (2006) *Desarrollo de laboratorios virtuales, interactivos y remotos utilizando Easy Java Simulations y Modelos Simulink*, XII Latin-American Congress on Automatic Control. Bahía, Brasil.

- S. Dormido, F. Esquembre, G. Farias, J. Sánchez (2005) *Adding interactivity to existing Simulink models using Easy Java Simulations*, 44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC'05) Seville, Spain.

Journal papers obtained in collaboration

The following papers are the result of collaboration with different groups. Only a selected number of works are presented. The main topic in these articles is pattern recognition

applied to the experimental fusion reactor TJ-II of CIEMAT.

- S. Dormido-Canto, G. Farias, J. Vega, R. Dormido, J. Sánchez, N. Duro, H. Vargas, A. Murari, and JET-EFDA Contributors. (2008) *Classifier based on support vector machine for JET plasma configurations*, Review of Scientific Instruments, Vol. 79, pp: 10F326-1/10F326-3, ISSN: 0034-6748.

- S. Dormido-Canto, G. Farias, R. Dormido, J. Sánchez, N. Duro, H. Vargas, J. Vega, G. Ratta, A. Pereira, A. Portas (2008) *Structural pattern recognition methods based on string comparison for fusion database*, Fusion Engineering and Design, Vol. 83, Issue 2-3, pp: 421-424. ISSN: 0920-3796. Ed. Elsevier.

- G. Farias, S. Dormido-Canto, J. Vega, J. Sánchez, N. Duro, R. Dormido, M. Ochando, M. Santos, G. Pajares (2006) *Searching for patterns in TJ-II time evolution signals*, Fusion Engineering and Design, Vol. 81, pp: 1993-1997, ISSN: 0920-3796, Ed. Elsevier.

- G. Farias, R. Dormido, M. Santos, N. Duro (2005) *Image Classifier for the TJ-II Thomson Scattering Diagnostic: Evaluation with a Feed Forward Neural Network*, Lecture Notes in Computer Science. Springer-Verlag, Vol. 3562, Part 2, pp: 604-612, ISSN: 0302-9743.

1.3.3 Research projects

The PhD thesis have been developed under the following research projects:

- Herramientas interactivas para el modelado, visualización, simulación y control de sistemas dinámicos (Interactive tools for modelling, visualization, simulation and control of dynamic systems). Reference: DPI 2004-01804.

- Control de sistemas complejos en logística y producción de bienes y servicios (Control of complex systems in logistic and production of goods and services). Reference: S-0505/DPI/0391.

- Modelado, simulación y control basado en eventos (Event-based modelling, simulation, and control). Reference: DPI2007-61068.

The main tasks performed in these projects are the following:

- Development of virtual laboratories for control teaching.
- Definition of a homogeneous approach to add interactivity and visualization to engineering simulation.
- Development of the JIMC library and the Jim server program.
- Implementation in Easy Java Simulations of a local and remote link with MATLAB/Simulink.
- Implementation in Easy Java Simulations of a local link with Scilab and Sysquake.
- Development of an approach to build networked control laboratories with MATLAB.
- Development of real-time control systems virtual laboratories using TrueTime and Easy Java Simulations.
- Implementation of the Java TrueTime (JTT) library to ease the creation of educational simulations of real-time control systems.
- Definition and implementation of an experimentation environment for simulations.

The collaboration with other research groups implied the following activities:

- MATLAB programming of custom pattern recognition tools.
- Application of support vector machines and wavelet transform for pattern recognition.
- Structural pattern recognition applied to the search of specific waveforms.
- Image classification in scattering Thompson diagnostic.
- Detection of different anomalies in atomic fusion signals.

Chapter 2

Design of Interactive Interfaces for Engineering Education

Information and communication technologies provide great new opportunities for education. In the last years, the scientific community has devoted great efforts to taking advantage of these new possibilities, applying them to various fields of engineering education, including control engineering (Heck 1999, Dormido 2002). Among all the opportunities that these technologies offer, three present features of special interest to teaching engineering: *network communications*, *visualization*, and *interactivity*.

The connectivity between many different computational devices is increasing every year. Networks such as the Internet are widely distributed in society, connecting people from almost any point of the globe. This situation gives educators the opportunity to offer their students new ways to access learning resources without time and location constraints.

On the other hand, visualization and interactivity have proved to be crucial aspects when designing simulations that are to be used for pedagogical purposes in the field of control engineering. The graphical capabilities of computers, using images or animations, can help students to understand more easily the key concepts of the system under study. Moreover, interactivity allows students to simultaneously see the response of the systems to any change introduced by the user (Dormido, Dormido-Canto, Dormido, Sánchez & Duro 2005, Sánchez et al. 2005). These features add to engineering simulations rich visual content and the possibility of an immediate observation of system response, which turns a virtual laboratory into a natural and human-friendly way to

learn, helping the student to get useful practical insight into control systems fundamentals.

Despite the educational importance of the three mentioned features, most of the engineering software currently used by teachers and students is far from the mentioned recommended way to teach and learn. Software for technical computing comes full of toolboxes for the design and rapid-prototyping of an engineering system, but it frequently lacks tools to facilitate adding user interaction and rich visualization capabilities to the simulations.

Another problem arises when an educator finally achieves (sometimes with great effort) to build an interactive simulation by using the facilities of a particular engineering software. He/She then realizes that the graphical user interface thus created can not be used with a similar simulation developed using a different engineering software, which is somewhat frustrating. User interfaces of standard simulations usually display common features, and it is reasonable to assume that they should be reused, independently of the software selected to create the model of the engineering simulation.

The purpose of this chapter is to describe a novel approach that defines a simple, but at the same time generic, protocol to help add human interfaces to engineering simulations. This approach splits the development of an interactive simulation for engineering in two separate tasks. On the one hand, the model of the engineering simulation is created using a standard simulation software for engineering. On the other hand, the interactive graphical user interface of the simulation is created using any high-level computer language or software tool specialized in the design and implementation of graphical user interfaces and which benefits from the graphical capabilities of modern computers. Finally, these two separate components are connected using a communication protocol in a clean, effective, and reusable way. The proposed approach is a generalization of previous work developed at the Department of Computer Science and Automatic Control in the UNED. Seminal ideas about the necessity to separate the model and the user interface, and about the manipulation of MATLAB (an engineering software) from a Java application (an user interface) can be found in (Sánchez et al. 2002, 2004, 2005).

The chapter starts with a short description of the current state of the art of the creation of engineering simulations, followed by a discussion of why interactivity and visualization are important in the field of engineering education. Then the *interoperate*

approach is introduced, the proposal of this thesis for the design of simulations that will help add human interfaces to engineering software. The chapter continues with a detailed description of the protocols that have been defined to support the interoperate approach, together with sample code that exemplifies their use. These protocols present variants according to how the model and the user interface are connected, either in local or remote form and, in this latter case, either synchronously or asynchronously.

2.1 Simulations with standard engineering tools

There are currently many tools that can help build a simulation of a large class of systems in control engineering. Most of these tools are extraordinarily useful to study the behaviour of the systems under different scenarios. For that reason, these tools are widely used by instructors to teach the main aspects of a particular control system. Nowadays MATLAB is the *de facto* standard software tool in control engineering.

MATLAB is a technical and numerical computing environment. The software provides a high-level programming language, which belongs to the so-called *fourth generation* software. Developed by The MathWorks, MATLAB allows matrix manipulation, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs in other languages. The capabilities of the environment are extended by a family of add-on application-specific solutions called toolboxes. These toolboxes, similarly to packages or libraries, allow users to apply advanced solutions to typical problems of a specific engineering area. Further information about MATLAB features can be found in Chapter 3.

The MATLAB environment allows the development of advanced graphical user interfaces using the set of tools named GUIDE. These tools greatly simplify the process of designing and building user interfaces. GUIDE can be used to lay out the graphical user interface using the GUIDE Layout Editor, the creation of the interface is done easily by clicking and dragging visual components (such as panels, buttons, text fields, sliders, and menus) into the layout area. GUIDE automatically generates an M-file (i.e., a file with MATLAB code) that controls how the user interface operates. Following an *event-based programming* paradigm (Faison 2006) (such as that used in languages like Visual Basic), authors can use the properties of each visual component to add callbacks in order to control the flow program by executing specific M-files when end users click

on a corresponding component of the graphical user interface.

Figure 2.1 shows a graphical user interface created with the GUIDE feature of MATLAB. The application performs image classification for a Scattering Thomson diagnostic employing very sophisticated algorithms like wavelet transform (Mallat 2001) and neuronal network (Hilera & Martínez 1995). The transform and the neuronal network are applied by means of executing some built-in functions of the *Wavelet* (Misiti et al. 2009) and *Neuronal Networks* (Demuth et al. 2009) toolboxes. Among other functions, the methods `appcoef2` and `detcoef2` were used here to perform the two-dimensional wavelet transform, and the `newff` method is used to create and perform a feed-forward backpropagation network. The wavelet transform strongly reduces the dimensionality of the original image (up to less than 1%), extracting the main features of the signal and eliminating the unnecessary noise. Once the image is processed, it is entered into the neuronal network to provide a classification of the signal in one of five possible classes.

This application allows the design of many kind of classifiers by modifying the wavelet transform and neuronal network parameters. The different classifiers designed offer several classification performance (i.e., percentage of correct classifications). The classifier can be used by students of an introductory pattern recognition (Duda et al. 2001) course to observe the effect of wavelet transform and neuronal network parameters over the classification (Farias, Dormido, Santos & Duro 2005, Farias, Santos & Dormido-Canto 2005, Farias et al. n.d.).

Listing 2.1 presents part of the M-file (generated by GUIDE) of the image classifier. The main visual component is a *figure* type (named `h0` in the code), which creates the frame where other components can be added. The creation of two other components: a push button (named `h1`) and a list box (named `h2`) can also be observed in the code. The push button represents the button with the caption *classify* in Figure 2.1, and performs the image classification by executing the M-file `classify.m` defined in its callback property. The list box allows end users to plot a specific image from the list of signals named *Data* in Figure 2.1. When a signal is selected from the list, the M-file `paintdata.m` is executed in order to show the corresponding image in Figure 2.1.

The image classifier is only a simple example of how engineering community can use, for researching or educational purposes, some of the state-of-the-art algorithms that MATLAB provides. Other similar examples of the use of GUIDE and MATLAB tool-

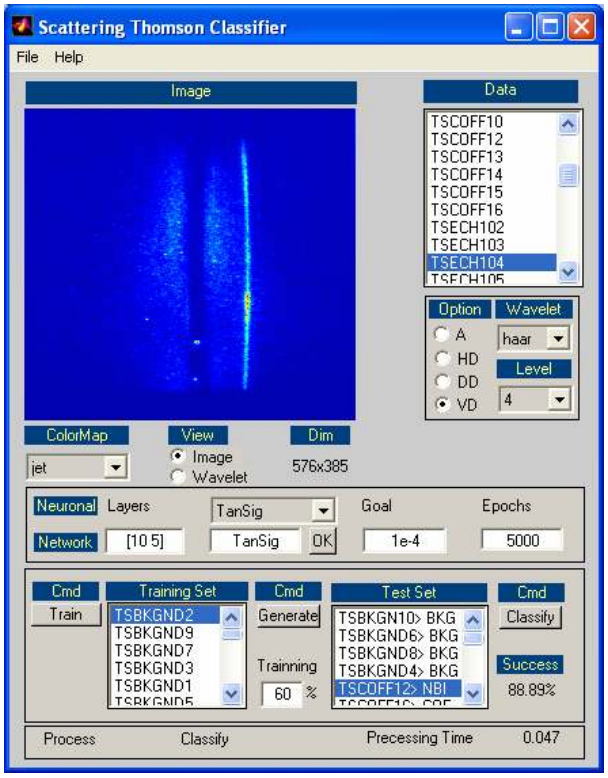


Figure 2.1: Image classifier GUI built in MATLAB.

boxes for pattern recognition can be found in (Farias, Dormido-Canto, Vega, Sánchez, Duro, Dormido, Ochando, Santos & Pajares 2006, Dormido-Canto, Farias, Dormido, Sánchez, Duro, Vargas, Vega, Ratta, Pereira & Portas 2008, Dormido-Canto, Farias, Vega, Dormido, Sánchez, Duro, Vargas, Murari & Contributors 2008, Vega et al. 2005).

Although instructors can use GUIDE to add interfaces to their engineering simulations, this is not the common situation on many areas of engineering. Maybe the main reason is that GUIDE helps to build easy simple control panels to manipulate a *batch* or static simulation. However, dynamic simulations require more advanced mechanisms to take into account any user interaction while the simulation is being performed. The implementation of these mechanisms requires a greater effort from the programming point of view.

As a consequence, the creation of graphical user interfaces finally requires some programming skills; this is specially true when the user interface includes many visual components.

With the exception of this programming barrier, MATLAB also provides many other sophisticated tools that do not require GUIDE to obtain a minimal graphical user interface. For example, Simulink can be used to simulate dynamic systems by connecting

pre-defined blocks of this MATLAB toolbox. The input and output of the system can be controlled by adding the suitable blocks to the Simulink model.

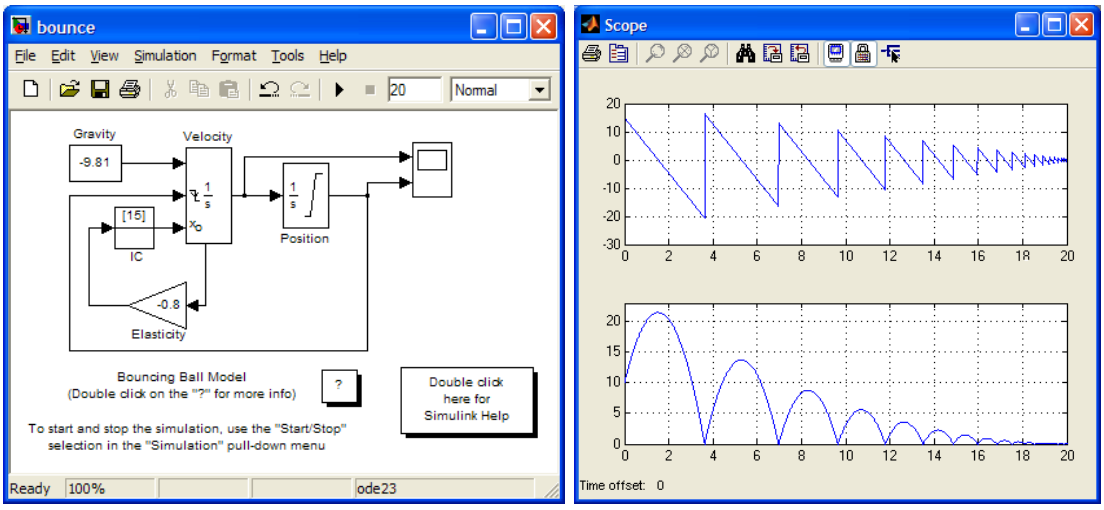
A typical example of a simulation developed in Simulink is displayed in Figure 2.2. The Simulink model (Figure 2.2a) simulates a non elastic bouncing ball under the effects of gravity.

The simulation of the bouncing ball works fine, and the plots of the speed and the position of the ball (Figure 2.2b) show clearly that the ball:

- Starts from a height of 10 meters with an initial speed of 15 meters per second.
- At the beginning goes up, reaching the maximum height at about 1.8 seconds.
- Stops bouncing after 20 seconds.
- Switches its speed from a negative to positive magnitude every time the ball hits the floor.

```
function fig = stneuronal()
load stneuronal.mat
h0 = figure('Color',[0.8 0.8 0.8], ...
        'Colormap',mat0, ...
        'FileName','stneuronal.m', ...
        'MenuBar','none', ...
        'Name','Scattering Thomson Classifier', ...
        'NumberTitle','off', ...
        'PaperPosition',[18 180 576 432], ...
        'PaperUnits','points', ...
        'Position',[298 129 434 513], ...
        'Tag','Fig1', ...
        'ToolBar','none');
...
h1 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[0.83 0.81 0.78], ...
        'Callback','classify.m', ...
        'Enable','off', ...
        'ListboxTop',0, ...
        'Position',[270.75 76.5 34.5 12.75], ...
        'String','Classify', ...
        'Tag','PushButtonClassify');
...
h2 = uicontrol('Parent',h0, ...
        'Units','points', ...
        'BackgroundColor',[1 1 1], ...
        'Callback','paintdata.m', ...
        'Position',[228.75 265.5 88.5 99], ...
        'String',mat5, ...
        'Style','listbox', ...
        'Tag','ListBoxData', ...
        'UserData',mat6, ...
        'Value',24);
...
```

Listing 2.1: Code of the image classifier GUI.



a) b)

Figure 2.2: A typical simulation with low level of interaction and visualization. a) Simulink model, b) Plots of the speed and vertical position of the ball.

- Loses energy after each bounce.

The common way to learn using this simulation is that users, maybe beginner engineering students, analyse the bouncing ball model by simply modifying the block parameters and running the simulation as many times as they need to.

Inspired by MATLAB, many other engineering tools have appeared in the last years. Many of these tools have a programming language compatible with MATLAB such as Sysquake (Calerga 2010) or the open source solutions Scilab (Scilab Consortium 2010) and Octave (Eaton 2002, 2010). However these tools do not offer all the functionality that MATLAB provides. For example, neither Scilab nor Octave have a feature similar to GUIDE that allows authors to generate automatically graphical user interfaces. Moreover, Sysquake offers a nice way to create interactive user interfaces, but does not provide the wide set of libraries included in powerful MATLAB toolboxes. This situation makes MATLAB the preferred tool in many scientific communities such as control engineering.

2.1.1 The importance of interactivity and visualization

The process of construction of simulations using tools such as Simulink can be easy from an authoring point of view. However, from a pedagogical point of view, to understand the response of the systems to any input changes is not trivial for beginner students. In engineering a typical system response analysis comes from the characteristics of output

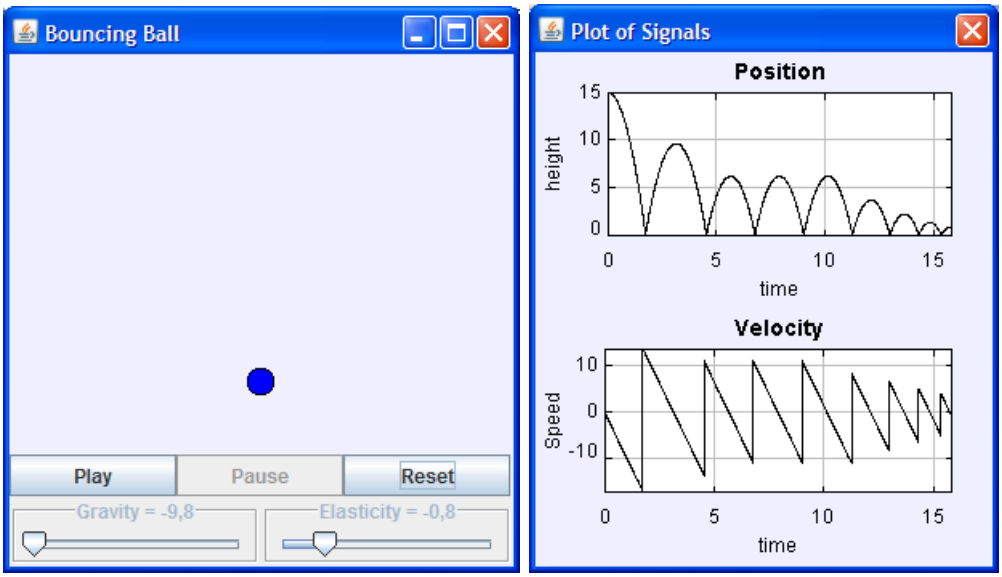
signals: the waveform, the period, the amplitude, etc. Therefore, the analysis of the system response is neither direct nor intuitive, because output signals are not very *human-readable*, which means that it is not intuitive to understand the input/output relationship of a system by observing plots of signals only. Hence, instead of using just signal plots, teachers need to add to their simulations a rich level of graphical content such as animations, images, two- or three-dimensional graphics, etc., in order to produce more intuitive and natural learning objects.

As a second important aspect of the learning process, many of the analyses are done *off-line*. That is, the signals can only be observed in full when the simulation has finished, with students rarely interacting with the system, changing parameters or inputs, while the simulation runs. However, recent research has shown that, especially in control engineering, interactivity is a fundamental element in the learning process (Dormido 2002, Dormido, Dormido-Canto, Dormido, Sánchez & Duro 2005, Sánchez et al. 2005, 2004).

The simulation displayed in Figure 2.2 models very well the system under study. But it clearly lacks a sufficiently rich level of interaction and visualization. In particular, there is no way to interact with the model while the simulation is running. Hence, if a student wants to interact with the simulation to see what happens if the gravity is decreased, or if the elastic coefficient of the ball is increased, or if the ball is moved to another position after 5 seconds, etc., he/she will need to stop the model, change the parameters, run the simulation again, and wait until its end to analyze the output data. This situation introduces unnecessary interruptions in the study of the input/output effects of the variables, which finally delays and encumbers the learning process of the system.

An alternative simulation of the bouncing ball with a higher degree of interactivity and visualization is displayed in Figure 2.3. In Figure 2.3a, a more suitable graphical user interface (GUI) is displayed. This GUI shows a ball (the blue particle) bouncing on the floor. The parameters, such as gravity and elasticity coefficient, can be modified by using sliders, and in addition, the vertical position of the ball can be changed by dragging up or down the particle in run time. Observing the plots in Figure 2.3b, it is easy to appreciate the effect of the change of the elasticity coefficient of the ball at t=5s to 1, and at 10s back to the previous value of -0.8.

This level of interactivity and visualization allows students to interact with the simulation while it is running, which provides them with a more natural way to study the bouncing ball system. This kind of simulations reduces significantly the learning curve of engineering systems.



a) b)

Figure 2.3: A simulation of a bouncing ball with higher level of interaction and visualization. a) Graphical user interface, b) Plots of the speed and vertical position of the ball.

The two simulations of the bouncing ball model represent two different paradigms to learn the behaviour of systems in engineering (Guzmán 2006). The differences between them are shown in Figure 2.4.

The non interactive paradigm splits the learning process into two sequential phases: analysis and synthesis. On the contrary, in the interactive paradigm, analysis and synthesis are done in the same phase, which, compared to the first paradigm, triggers

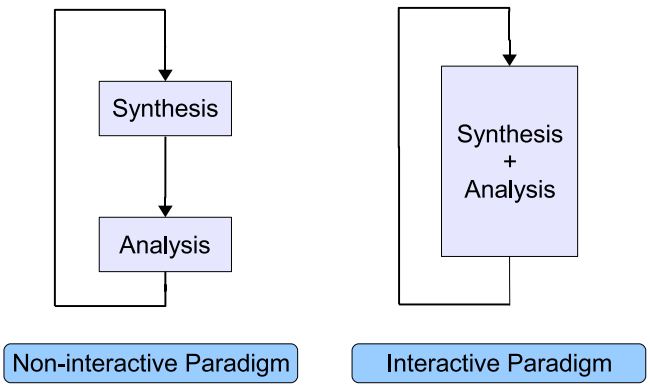


Figure 2.4: The two learning paradigms to study the behaviour of a system.

more easily the analysis of the process in the student's mind. This faster comprehension reduces the time required to understand some input/output relations, and therefore, to learn the process behaviour of the global system.

In opposition to the off-line simulation, this thesis favours the *on-line* (or *on-the-fly*) interactive way of learning, which allows students to modify system inputs or parameters while the simulation is running. This interactive process helps students better understand the input/output relations and appreciate or evaluate the degree of influence that any input has in the global response of the system.

2.2 The interoperate approach

The alternative simulation shown in Figure 2.3 was created reusing the Simulink model of Figure 2.2a, by adding to it a graphical user interface created in Java, a high-level programming language that offers many advantages for the creation of sophisticated user interfaces.

In this thesis, the claim is that this combined use of different tools, each suited best for a given task (engineering software specialized in modelling engineering systems for the model of the simulation, and authoring tools suited for the design and implementation of user interfaces), makes a perfect combination that brings the best of both worlds to educators in order to develop their learning simulations. This will be the preferred approach in this thesis, to reuse existing models of simulations, created using standard engineering software, by adding a human interface with a high level of interactivity and visualization to produce a complete, teaching-efficient virtual laboratory.

Alternatively, instructors can always try to add the required interactivity and visualization to an existing engineering simulation using the same engineering software they used to create the model (e.g. using GUIDE in MATLAB). However, this approach usually has the following shortcomings:

- An interactive simulation created with a given engineering software requires that the engineering software be installed on the user's computer.

- Different engineering software require different ways to create similar user interfaces.

- Engineering software does not always provide tools to ease the building of interactive simulations.

On the contrary, since the interoperate approach splits the interactive simulation into two components, instructors can take advantage of high-level programming languages to develop the user interface independently of the engineering software used. Instructors can even use specialized authoring tools (such as Easy Java Simulations (Esquembre 2010), see Chapter 4 for further information) to facilitate the creation of the user interface of the interactive simulation. This approach also facilitates the reuse of components, either using the same engineering simulation with a more elaborate graphical user interface or using the same user interface with a simulation described in a different engineering software. Finally, the approach can benefit from network communications to interoperate the human interface and the engineering software on different computers, eliminating the need to install the engineering software on the student's computer.

After this analysis, an exact definition of the interoperate approach is now provided:

The *Interoperate Approach* is a uniform and effective way to create interactive engineering simulations by manipulating the engineering software from an interactive human interface.

The key concept of the interoperate approach is *interoperability*, which is the ability of two or more systems or components to exchange information and to use the information that has been exchanged (IEEE 1990). The two systems that exchange information are the engineering software and the human interface.

A scheme of the interoperate approach is shown in Figure 2.5. The human interface is created using a high level computational language to build the graphical user interface of the interactive simulation, and it is later connected to the existing (non-interactive) model of the engineering simulation using a standardized communication protocol (see Section 2.3) that controls the engineering application.

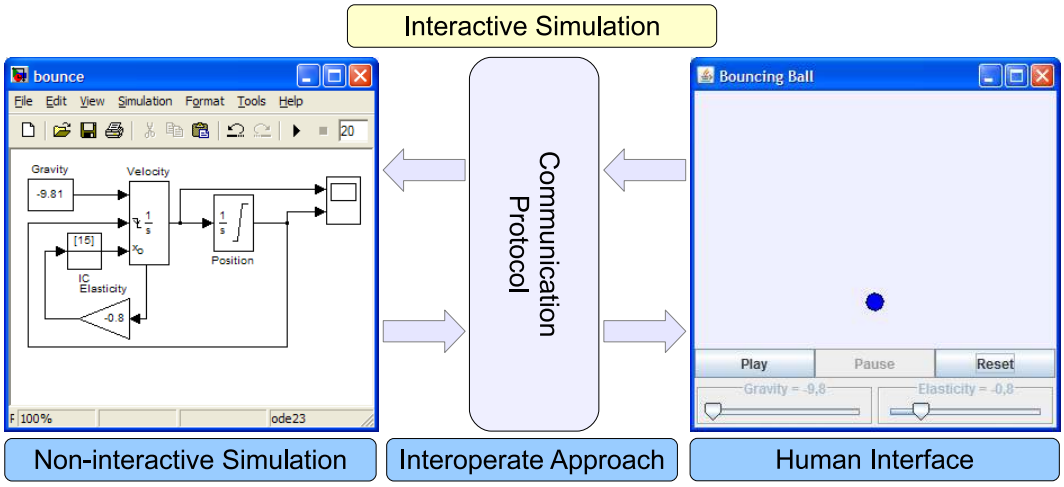


Figure 2.5: Adding a human interface to an existing, non interactive, engineering simulation.

2.2.1 Adding interactivity and visualization

The graphical user interface of the interactive simulation shown on the right in Figure 2.5 can be created using any high level computer language. In this thesis, Java has been chosen as the implementation language of the proposed approach. The main reason is that Java is currently one of the most popular programming languages, addressing not only the design of graphical interfaces, but also a huge range of applications such as applets, network communications, image, video, and sound processing, 3D graphics manipulation, physics and mathematics, interoperation, mobile programming, etc. This popularity is especially true in the educational world, which benefits noticeably from the pedagogical advantages of the use of computer simulations in the learning process (Heck 1999, Dormido 2002, Dormido, Dormido-Canto, Dormido, Sánchez & Duro 2005).

Even with the use of a flexible programming language such as Java, creating advanced visualizations and interactive user interfaces to be used with engineering simulations is not an easy task, specially for teachers who are not experts in computer programming. It is actually a time consuming task for even advanced programmers. Fortunately, there are utility libraries and specialized authoring tools that enormously facilitate the task. In this thesis, the Easy Java Simulations (EJS) authoring tool has been chosen, because with this software tool authors do not need advanced programming skills to produce virtual labs with a high level of interactivity and visualization. Since EJS is a Java-based program, it can easily use the Java implementation of the communication protocol provided in this work. Furthermore, the thesis has even contributed to the

development of EJS, embedding the protocol into it, so that it naturally helps implement the interoperate approach (see Chapter 4).

2.3 Defining a communication protocol

Conceptually, in any interactive simulation that uses the model of an existing engineering software, two separate components that interoperate can be distinguished: the *client application* and the *external application*.

The client application represents the human interface, i.e., the computer program or graphical interface that the user observes and manipulates. The external application runs the engineering simulation, that is, the simulation created with the standard engineering software that models the process of interest. The external application is controlled by the client application and provides it data for visualization. The client application adds to the underlying (external) engineering application an upper layer that allows students to manipulate and visualize the system's response interactively.

For this scheme to work correctly, both client and external applications must exchange data continuously as the simulation is running. And they do so for a number of tasks. For example, the external application should be correctly configured and launched at start-up. It should also provide a way for the client to read, in run-time, the value of variables of the model, for visualization. The external application should also allow the client to start the simulation, pause it to modify any parameter or state variable of the system, and resume the simulation, as required by the user.

For its part, the client should be configured to control, query, and send data to the external application using a protocol that the external application can understand, without interrupting the execution of the external application and keeping perfect synchronization between the visualization offered to the user and the model of the external application.

As part of the work of this thesis, different prototypes that implemented the interoperate approach have been extensively tested, until a standard, coherent definition of the required communication protocol was finally established. The following protocol represents a simple, but powerful application programming interface (API) that any engineering software must conform to, in order to provide all the features required to effectively implement the interoperate approach.

The protocol comes in two levels: low and high. A *low-level protocol* has been defined that lists all the required communication mechanisms that will offer the client application a complete and flexible control of the external application. Any external application must follow this low-level protocol in order to successfully implement the interoperate approach. But a *high-level protocol* has also been defined. This high-level protocol offers a simplified list of communication instructions that can, in most practical circumstances, help authors successfully implement the interoperate approach with a minimum of programming effort. The high-level protocol is based on the low-level protocol, in the sense that it ultimately uses the API provided by the external application, and offers the client application the minimum, but sufficient, set of rules to manipulate the exchange of data.

This two-level communication protocol scheme allows authors the choice to manipulate the engineering simulation in two different ways. On one hand, the high level protocol gives authors the control of the non-interactive simulation at a simple, high level of abstraction, hiding a number of details, but still providing an effective link between the non-interactive model and the interactive user interface of the simulation. On the other hand, the low level protocol gives authors total control of the external simulation, providing an enhanced link between both simulations, bringing a higher level of interaction and visualization.

Usually, the high level protocol is all that most authors will need to fulfill their interaction requirements, it being therefore the recommended entry level for authors who are not expert programmers or do not need a very detailed control of the communication between client and external applications. The low level protocol is the preferred choice for authors that need full control of the original simulation and the communication mechanism. Needless to say, a correct use of the low level protocol to design an interactive simulation requires some more programming effort than that of the high level protocol. But it should also be noted that it can also result in a more efficient final application (in terms of communication traffic and execution times).

Because Java was chosen as the programming language to create the graphical user interfaces, this section uses also Java to exemplify the implementation of the communication protocol and to provide some sample code with examples of use. The choice of this well-known language allows to make the protocol explicit in a way that any pro-

grammer of a different language can understand and adapt. It is important to note that there is in principle nothing in this Java implementation that prevents the communication protocol to be implemented using a different programming language. However, the use of any particular programming language in a computer implementation always leaves traces of a peculiar accent. In what follows, any special Java feature that may have been used in the implementation that would require a special adaptation to other languages will be explicitly mentioned.

2.3.1 The Java language

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

Among others, Java provides many interesting features such as:

Platform Independence The Write-Once-Run-Anywhere ideal has been achieved for almost all platforms. Java programs can in general be executed without any problems in different operating systems such as Windows, Solaris, and Linux. Even more, there are many small Java applications (normally games) running today in mobile phones.

Object Oriented Object oriented throughout - no coding outside of class definitions, including the `main()` method. An extensive class library in the core language packages is available.

Compiler/Interpreter Combo Code is compiled to bytecodes that are interpreted by a Java virtual machine (JVM). This provides portability to any machine for which a virtual machine has been written. The two steps of compilation and interpretation allow for extensive code checking and improved security.

Robust Exception handling built-in, strong type checking (that is, all data must be declared an explicit type), local variables must be initialized.

Security No memory pointers, no preprocessor, and array index limit checking. Programs run inside the virtual machine sandbox.

Automatic Memory Management Automatic garbage collection - memory management handled by the JVM.

Good Performance Interpretation of bytecodes slowed performance in early versions, but advanced virtual machines with adaptive and just-in-time compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs.

Threading Lightweight processes, called threads, can easily be spun off to perform multiprocessing. Java can take advantage of multiprocessors where available.

Built-in Networking Java was designed with networking in mind and comes with many classes to develop sophisticated Internet communications.

This extensive list of good attributes justifies the selection of Java as the language for the implementation of the algorithms and protocols of this thesis. This decision is also suggested by the high popularity of Java in the computer and Internet. This is specially true in the educational world, which is benefiting noticeably from the pedagogical advantages of the use of computer simulations in the learning process (Sánchez et al. 2002, Vormoor 2001, Piguet & Gillet 1999, Yen et al. 2003, Balestrino et al. 2009).

2.3.2 Low-level protocol

To manipulate and visualize the system's response interactively, client and external applications must keep a continuous flow of information composed of data and commands. The data is provided by the external application as the simulation runs. The commands are used by the client application to control the execution of and query the external application.

The first command required in this communication flow is a request for connection. Some engineering programs may require an initialization command, for instance when some code has to be executed in order to prepare the engineering simulation before it is actually run. Once the connection is established, the external application is manipulated by the client application according to its logic, executing commands, retrieving information about the results of the executed commands (which can be very useful to fix problems found during the simulation), and setting and getting the value of variables.

There are, therefore, three blocks of communication actions an external application must support:

Connection and configuration These are actions that initialize the external application, prepare it for operation, and eventually quit it in an orderly way.

Setting and getting values These actions allow the client to set and get the value of any (accessible) variable of the external application.

Control commands These actions control the execution of the external application from the client.

These actions are supported by a number of methods, as defined by the Java *interface* `ExternalApp`. In the rest of this subsection, this interface is analysed in detail.

Connection and configuration methods

The methods of the `ExternalApp` interface that make the connection and configuration block are listed in Listing 2.2

```
1  /**
2   * Starts the connection with the external application
3   * @return boolean true if the connection was successful
4   */
5  public boolean connect();
6
7  /**
8   * Finishes the connection with the external application
9   */
10 public void disconnect();
11
12 /**
13  * Checks if the connection was successful
14  * @return boolean true if the connection was successful
15  */
16 public boolean isConnected();
17
18
19 /**
20  * Accepts an initialization command to use whenever the system is reset
21  * @param command String
22  */
23 public void setInitCommand (String command);
24
25 /**
26  * Gets the initialization command
27  * @return String the initial command
28  */
29 public String getInitCommand();
```

Listing 2.2: Connection and configuration methods of the `ExternalApp` interface.

Setting and getting values

The `ExternalApp` interface includes the methods displayed in Listing 2.3 to set and read the values of variables of the external application. Notice that variables are identified in the external application by a unique name. The external application will take care of the internal work required to connect this name to the actual variable. Not all variables of an external application need to be accessible through these methods.

```
1  /**
2   * Sets the value of the given variable of the application
3   * @param variable String the variable name
4   * @param value String the desired value
5   */
6  public void setValue(String variable, String value);
7
8  /**
9   * Gets the value of a String variable of the application
10  * @param variable String the variable name
11  * @return String the value
12  */
13 public String getString (String variable);
14
15 /**
16  * Sets the value of the given variable of the application
17  * @param variable String the variable name
18  * @param value double the desired value
19  */
20 public void setValue(String variable, double value);
21
22 /**
23  * Gets the value of a double variable of the application
24  * @param variable String the variable name
25  * @return double the value
26  */
27 public double getDouble (String variable);
28
29 /**
30  * Sets the value of the given variable of the application
31  * @param variable String the variable name
32  * @param value double[] the desired value
33  */
34 public void setValue(String variable, double[] value);
35
36 /**
37  * Gets the value of a double[] variable of the application
38  * @param variable String the variable name
39  * @return double the value
40  */
41 public double[] getDoubleArray (String variable);
42
43 /**
44  * Sets the value of the given variable of the application
45  * @param variable String the variable name
46  * @param value double[][] the desired value
47  */
48 public void setValue(String variable, double[][] value);
49
50 /**
51  * Gets the value of a double[][] variable of the application
52  * @param variable String the variable name
53  * @return double the value
54  */
55 public double[][] getDoubleArray2D (String variable);
```

Listing 2.3: Setters and getters methods of the `ExternalApp` interface.

Control commands

Finally, the `ExternalApp` interface contains a number of methods that allow the client application to send control commands to the external application. Listing 2.4 shows these methods.

```
/**
 * Evaluates a given command in the external application
 * @param command String to be executed
 */
public void eval (String command);

/**
 * Resets the application
 */
public void reset();

/**
 * The result of last action can be read using
 * this method.
 * @return String the result of last action
 */
public String getActionResult();
```

Listing 2.4: Control methods of the `ExternalApp` interface.

2.3.3 Sample use of the low-level protocol

The use of the interface is straightforward. Listing 2.5 shows an example of use of a given external application (`MyExternalApp`) that implements the `ExternalApp` interface.

```
import es.uned.dia.interoperate.ExternalApp;

public class LowLevelExample {
  public static void main (String[] args) {
    // Declare local variables
    double time=0, frequency=1, value=0;
    // Create the ExternalApplication
    ExternalApp externalApp = new MyExternalApp();
    // Start the connection
    if (!externalApp.connect()) {
      System.err.println ("ERROR: Could not connect!");
      return;
    }
    // Set the frequency
    externalApp.setValue("f",frequency);
    // Perform the simulation
    do {
      externalApp.setValue("t",time);
      externalApp.eval("y=sin(2*pi*f*t)*cos(t)");
      value=externalApp.getDouble("y");
      System.out.println("time:"+time+" value:"+value);
      time=time+0.1;
    } while (time<=10);
    // Finish the connection
    externalApp.disconnect();
  }
}
```

Listing 2.5: Sample code of use of the low-level protocol.

When run, the program produces the following output:

```
time:0,000 value: 0,000
time:0,100 value: 0,585
time:0,200 value: 0,932
time:0,300 value: 0,909
time:0,400 value: 0,541
time:0,500 value: 0,000
time:0,600 value:-0,485
time:0,700 value:-0,727
time:0,800 value:-0,663
time:0,900 value:-0,365
time:1,000 value:-0,000
...
```

The very simple sample code in Listing 2.5 above shows the basic operation required to control an external application. A more sophisticated example, which involves the use of the Easy Java Simulations for the visualization of results is displayed in Figure 2.6.

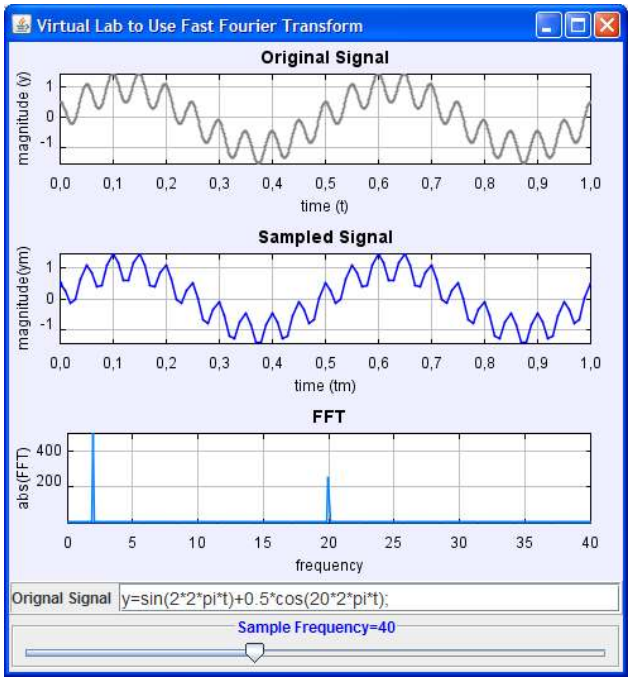


Figure 2.6: A virtual lab to perform Fast Fourier Transform. The three plots represent the signal, the sampled signal and the amplitude of the FFT.

In this example, the user enters a one-dimensional signal by indicating its analytic formula. This signal is sent to an engineering software (for instance MATLAB), which returns the computed Fast Fourier Transform (FFT) (Oppenheim et al. 1997). Both the original signal and the computed transformation are plotted in the user interface of the virtual lab. Users can also modify the frequency at which the original signal is sampled. The effect of this frequency on the processed signal and the computed FFT can be immediately observed by students. Note that the two frequencies ($2Hz$ and $20Hz$) of the signal:

$$y = \sin(2 \cdot 2\pi \cdot t) + 0.5 \cdot \cos(20 \cdot 2 \cdot \pi \cdot t)$$

are correctly identified by the computed FFT.

Other, more sophisticated examples, created with the help of Easy Java Simulations are shown in the next chapters.

2.3.4 High-level protocol

As the sample code above shows, the low-level protocol provides all that is needed by a programmer to control an external application. (Although extensions of the `ExternalApp` interface will be required for remote communication of client and external application, as discussed in Section 2.4.) However, as simulations grow in size and sophistication, it becomes more and more difficult for a not-so-expert programmer to keep control of all the method invocations required to keep the client and external applications perfectly synchronized.

For this reason, additional methods have been added to the `ExternalApp` interface that make use of the low-level protocol to provide a higher level protocol for instructors to set up an efficient communication scheme between client and external applications using a reduced set of instructions. The use of this high level protocol may make the final simulation slightly less efficient, but the gain in simplicity provides advantages in the form of shorter development times and also in the reduced possibility of introducing synchronization errors.

The concept of this utility high-level protocol comes from the identification of the basic mechanism of communication between client and external application. In a large percentage of simulations, the author just needs to connect and configure the external application and then make sure that a number of variables of the user interface are *linked* to corresponding variables of the external application. Interactive communication between both applications requires that any change in the user interface of the client application immediately be reflected by the external application and vice versa. That two variables are linked means that both must hold the same value, and that changes occurring in one of them must be automatically propagated to the other one. The high-level protocol implementing class must then assume the responsibility of this synchronization, not the author of the simulation.

The flow of information between the applications is needed because the system is dynamic, i.e., responds to changes as time increases. To control this situation, the high-

level protocol should provide a way to configure the simulation, which means to define *when* and *how* the simulated system has to respond to a change in time. In many cases, for instance, an engineering simulation consists of a set of ordinary differential equations (ODE), so that the way to simulate the system uses a suitable ODE solver (such as a Runge-Kutta method), and to control when to execute an integration step. In other cases, a simulation consists of a sequence of statements that have to be continuously executed by the external application to run the simulation. The protocol should take into account all those situations.

The protocol is conceived as an extension of the methods of the low-level protocol, and therefore offers all the previous functionality, and is included in the `ExternalApp` interface. The new methods added by the high-level protocol can be divided in two main groups: The linking methods and the control methods.

Linking variables

The first group of the additional methods of the high-level protocol are used to link variables of the client to variables of the external application. This linking process is done by name and is accomplished by using the methods in Listing 2.6.

```
1  /**
2   * Sets the client application
3   * @param clientApp Object the client application. Reflection is used to access
        the variables in the client.
4   */
5  public void setClient(Object clientApplication);
6
7  /**
8   * Links a client variable with a variable of the external application
9   * @param clientVariable String the client variable
10  * @param externalVariable String the external application variable
11  */
12 public boolean linkVariables(String clientVariable, String externalVariable);
13
14 /**
15  * Clears all linking between variables
16  */
17 public void clearLinks();
```

Listing 2.6: Methods to link variables of client and external applications.

This linking process uses the low-level protocol to access the variables of the external application and *reflection* to access the variables of the client. Reflection is a Java mechanism that allows one class to access public variables of another object; as it is the single Java artifact that has been used here, it may require a different programming implementation in a different language.

Control commands

The control of the external application is extended done through a reduced set of methods, listed in Listing 2.7.

```
/**
 * Some external applications, such as Matlab, can continuously execute
 * a command after every step().
 * @param command a String to be executed
 */
public void setCommand(String command);

/**
 * Gets the command to be executed by the external application.
 * @return String the command
 */
public String getCommand();

/**
 * Steps the application a given step or a number of times.
 * If getCommand() is non-null the command is executed that number of times.
 * If getCommand() is null, the dt parameter is passed down to the externall
     application,
 * and the actual meaning of this parameter dt will depend on the implementing
     class
 * @param dt double
 */
public void step (double dt);

/**
 * Synchronizes client and external applications
 */
public void synchronize();
```

Listing 2.7: Control methods of the high-level protocol.

A simple example will be used to discuss how the high-level protocol works.

2.3.5 Sample use of the high-level protocol

Listing 2.8 shows the high-level protocol in action. This is a version of the same program in Listing 2.5, but now using the high-level protocol.

```
import es.uned.dia.interoperate.ExternalApp;

public class HighLevelExample {
  // Declare local variables
  public double time=0, frequency=1, value=0;

  public static void main (String[] args) {
    new HighLevelExample();
  }

  public HighLevelExample () {
    // Create the external application
    ExternalApp externalApp = new MyExternalApp();
    // Set the client application
    externalApp.setClient(this);
    // Link variables with the external app's
    externalApp.linkVariables("time", "t");
    externalApp.linkVariables("frequency", "f");
    externalApp.linkVariables("value", "y");
    // Configure the external application
    externalApp.setCommand("y=sin(2*pi*f*t)*cos(t)");
    // Start the connection
```

```
23      if (!externalApp.connect()) {
24        System.err.println ("ERROR: Could not connect!");
25        return;
26      }
27      // Perform the simulation
28      do {
29        externalApp.step(1); // step once
30        System.out.println("time:"+time+" value:"+value);
31        time=time+0.1;
32      } while (time<=10);
33      // Finish the connection
34      externalApp.disconnect();
35    }
36  }
```

Listing 2.8: Sample code of use of the high-level protocol.

When the program is run, it produces the same following output as the previous example of Listing 2.5.

There are five blocks of code required to run a simulation using the high-level protocol: linking of variables, connection, configuration, stepping, and disconnection. The stepping phase is where the real work is done.

Once the client's variables are linked to the external variables, the connection is established and the external application is configured, successive calls to the `step` method handle automatically all required calls to methods of the low-level protocol. The `step` command performs sequentially three actions:

- Sets all external variables to the corresponding client values.
- Runs or executes the engineering simulation as many times as indicated by the `dt` parameter.
- Gets all external values and sets the corresponding client variables.

After the call to the `step` method, therefore, the variables of the client which have a link will hold the same values as those of the external application linked to them. The client can then use them for whatever visualization is required. If the client changes any of these variables (for instance, due to user interaction), the `step` method will take care of updating the external variables prior to stepping the external application.

Sometimes, changing a client's variable has implications to the external applications other than the mere change of the value of a variable. In these cases, a call to the `synchronize` method is required and it is the author's responsibility to include it in the program.

Figure 2.7 shows a diagram of the execution of an external application using the high-level protocol.

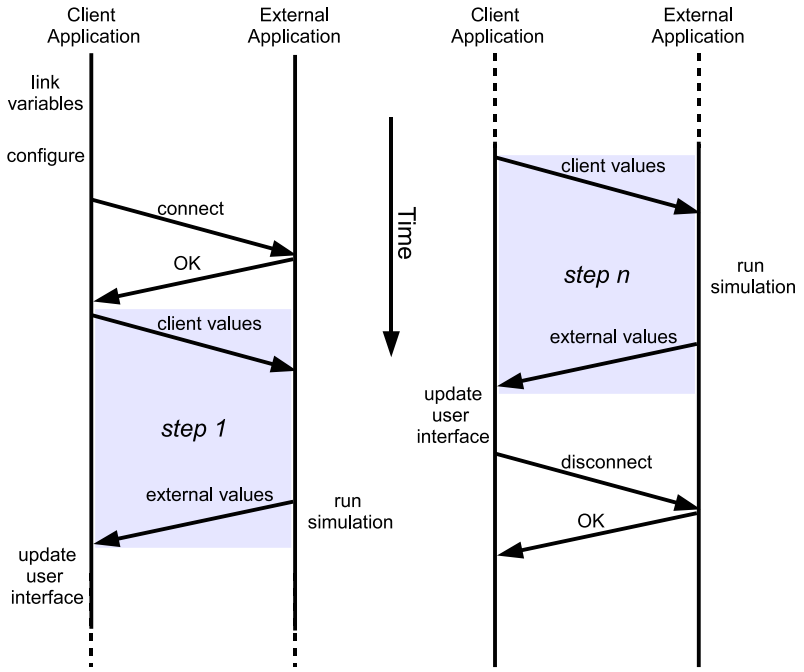


Figure 2.7: High-level protocol in action. There are five main phases in the execution of a simulation using the protocol: linking variables, configuration, connection, stepping and disconnection.

2.3.6 Low or high level protocol?

In a large number of simulations, the high-level protocol is all that is required to control interactively the original engineering simulation. However, there are situations where an interactive simulation requires more flexibility. For example, to execute some piece of code in the engineering software at a specific time, or when the simulation uses a large number of variables which are not changed very often. In these latter cases, the use of the low-level protocol is preferred, both to give to the user interface of the interactive simulation the possibility to control the engineering software in a more refined form than the high-level protocol permits, and also to improve performance, minimizing communication traffic.

When instructors face the design process of the interactive simulation following the interoperate approach, they have to decide what level of control of the engineering software is required. Normally the high-level protocol should be enough for developing a wide range of interactive simulations. A good idea is to start using the high-level protocol, and, only if it does not offer full control of all aspects of the interactive simulation,

partial or full use of the low-level protocol should be introduced.

Another application field where the low-level protocol may be preferred is the implementation of virtual labs of a non dynamic nature. This type of laboratories is used mainly to perform some given actions over collected data. For example, a typical application of the low-level protocol is the implementation of a virtual lab for signal processing displayed in Figure 2.1 above (or the image classifier shown in Figure 2.1). In this virtual lab students can calculate the Fast Fourier Transform (FFT) of a given signal.

2.4 Remote interaction of engineering software

Network communications are ubiquitous in modern life. This interconnection provides challenges and also opportunities to engineering education. Web-based courses are being increasingly used by educational institutions to support (or even a substitute) communication with students.

The interoperate approach can benefit from the network communications to connect the human interface and the engineering software. In other words, client and external applications can be located and run on different computers using the network to communicate. This feature can be used to offer access to limited resources to a wide audience of students.

The possibility of using a network connection between both applications introduces a new factor that differentiates between two types of connections: *local links* and *remote links*. If the client application and the external application are located on the same computer, then the connection is called a local link. This is the situation that has been described so far. If, on the contrary, the client application and the external application are located on two different computers and communicate through a network connection, then the connection is called a remote link. This section analyses this latter mode of operation of the interoperate approach.

2.4.1 Remote link

Performing engineering simulations over networks can be especially useful when hardware or software resources are limited. On some occasions, the simulation needs a high level of computational power, requires special hardware (such as hardware-in-the-loop

simulations (Obaidat & Papadimitriou 2003, Karayanakis 1995)), or the engineering software is not installed on the students' computers.

Instructors can then develop a simulation where all the computation required by the simulation (the model) can be run on a computer where the needed hardware and software resources are available, and at the other end, a computer at the student's desk is used to show the results of the simulation and offer user interaction. The two ends of this connection are normally named server and client sides, respectively. The network is used to connect both sides.

The interoperate approach also gives instructors the possibility of building these type of simulations by using a remote link. The only requirement is to solve the technical problem of providing a Java class that offers the protocol API that will handle the communication to an engineering simulation running in a separate computer. An example of how this challenge can be solved is described in detail in Chapter 3.

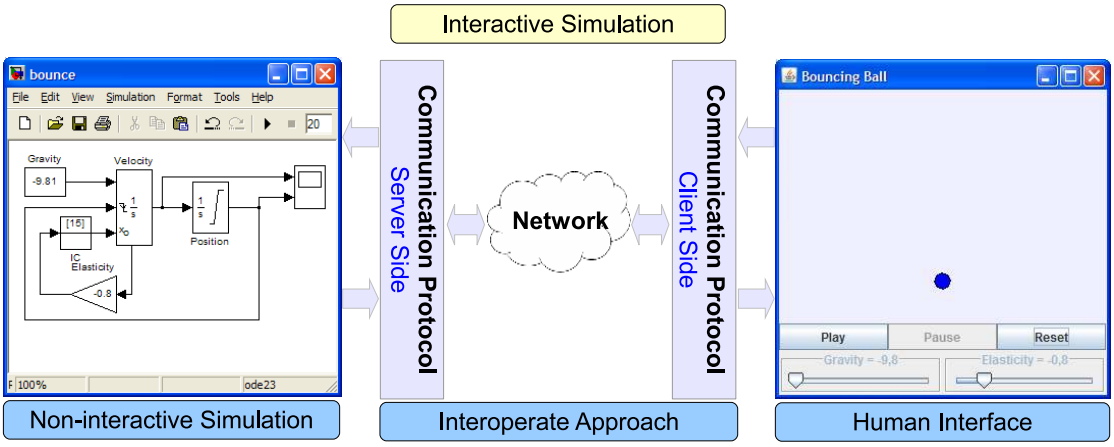


Figure 2.8: The interoperate approach for a remote link.

Once this technical problem is solved, the use of the communication protocol in a remote link is (in principle) straightforward. The scheme of such a remote link is presented in Figure 2.8. Notice that the communication protocol is implemented on both the client and server sides. The approach encapsulates the network in such a way that it becomes transparent to the end user. The only differences observed by end users, with respect to a a simulation using a local link, can be a slower performance caused by network delays. From the design point of view, the creation of the interactive simulations in both local and remote links is similar.

In practise, according to the experience obtained during the doctoral period, the

possibility of network delays introduces additional requirements for the protocol. It is true that, if network delays are negligible (such as in a fast intranet), authors can use the interoperate approach indistinguishably of the type of the link selected to deploy the interactive simulation. However, if network delays are noticeable, the simulation will give a poor performance. For this reason, and in order to minimize this undesired effect, a new version of the communication protocol was introduced. This new version is termed **asynchronous** to distinguish it from the standard one, that will be referred to as **synchronous**.

The introduction of asynchronous links has an effect only when authors use the high-level protocol, because the flow of information between client and external application will be accomplished differently depending on the version used. Because the low-level protocol provides authors with a direct control over the exchange of information among the applications, it is the responsibility of the programmer to decide how to handle this flow of information, taking into account or not the possible network delays. Automatic handling of these aspects is another feature that makes the high-level protocol attractive for not-so-expert programmers.

In order to explain the difference in the execution of synchronous and asynchronous links, Table 2.1 lists all individual phases of the stepping of a simulation using the remote link.

Table 2.1: Phases of the stepping operation in a remote link.

Phase	Action
1	The client side gets the values of the client variables.
2	The client side sends to the server side the values obtained.
3	The server side sets the corresponding variables of the external application.
4	The external application runs the engineering simulation once.
5	The server side gets the values of the external variables.
6	The server side sends these values back to the client side.
7	The client side sets the corresponding variables of the client application.
8	The client application updates the user interface.

Figure 2.9 shows the phases of Table 2.1 in a visual form. Note that all phases have to be executed independently of the version of the remote link selected. The order in which they are executed will establish the difference between both versions.

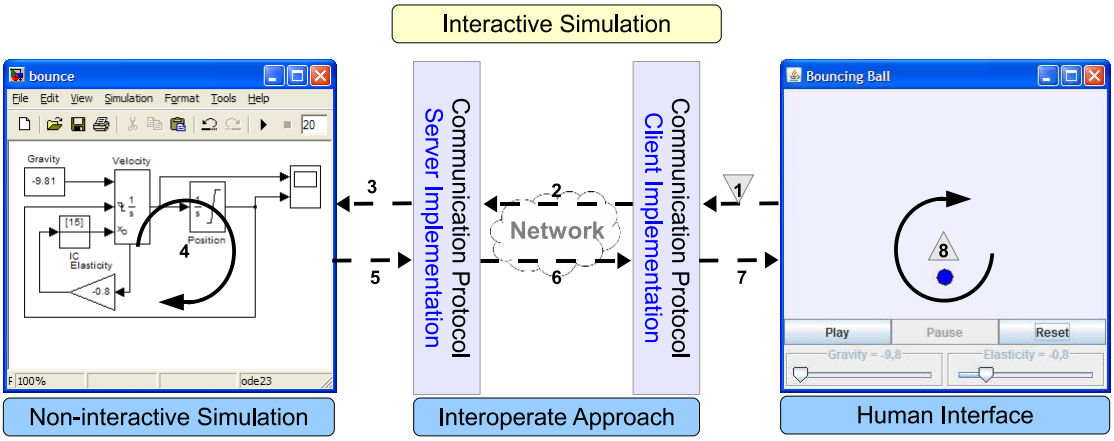


Figure 2.9: Phases of the interoperate approach for a remote link.

2.4.2 Synchronous remote link

The synchronous link between client and external application is bidirectional. By this, it means that in every remote simulation step, all phases of Table 2.1 are executed. Hence, information flows constantly in both directions.

Figure 2.10a shows a chronograph of the synchronous link in action. Note that, in the remote execution of a synchronous link, possible delays may be caused not only by the network, but also when the client or external application are computing. This delay can be more noticeable because when one of the applications is being executed, the other application must wait, even if the corresponding computer is idle. Therefore, the total delay in every remote execution step is the sum of the network delay and the processing time of both applications. However, network delays usually play the major part in observed delays when running a remote simulation, since network delays are, in general, longer than computing delays.

Despite the problem of an increased delay, the synchronous link presents some advantages, since it is similar to a local connection between client and external applications. It actually looks the same from the author's point of view. This means that a simulation created originally to work using a local link can be easily transformed into a simulation that uses a synchronous remote link. The single change required to transform the original simulation into a remote one is to indicate the network location of the computer where the engineering software is running.

For example, in the sample program of Listing 2.8, the only change required would be to replace the line:

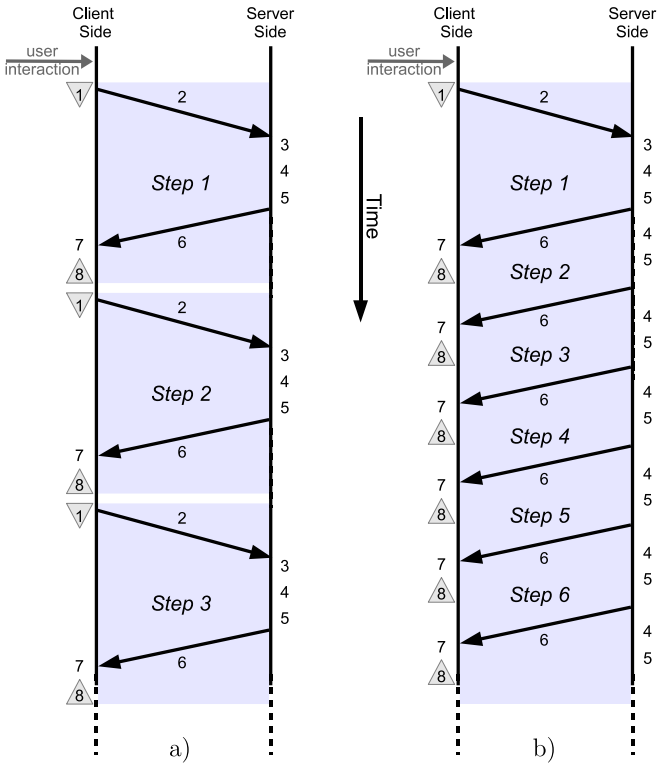


Figure 2.10: Remote links in action, a) Synchronous b) Asynchronous.

```
ExternalApp externalApp = new MatlabExternalApp();
```

with the following one:

```
ExternalApp externalApp = new RemoteMatlabExternalApp("configuration");
```

where the string `configuration` should indicate the network address of the external application.

2.4.3 Asynchronous remote link

The performance of a synchronous link is acceptable whenever network delays are negligible, such as in a local area network (LAN). However, in a wide area network (WAN), e.g. Internet, the evolution of the simulation could become sluggish. This is where the asynchronous link comes in handy, because this link has been developed to minimize the effects of network delays.

The synchronous link requires a huge amount of transaction between the client and external applications to exchange information when the remote simulation is running. Obviously, the more transactions are required, the slower is the simulation.

To reduce these execution delays, the asynchronous link intendedly does not keep a continuous synchronization between both applications. This carelessness is based on

the key idea that the end user does not interact with the simulation at every moment. Usually, the end user make changes to the simulation using its GUI only sporadically, spending most of the time precisely observing the system response after any modification in the parameters that govern the system.

Based on this idea, the communication between the applications is not bidirectional as in the case of the synchronous link, but *unidirectional* for most remote steps, showing bidirectionality only whenever the end user interacts with the view of the simulation.

Hence, most of the time, the external application will just be continuously running the engineering simulation, sending back the values of external variables to the client application. The client will spend most of its time just receiving and displaying those values. This situation will be interrupted only if and when the end user introduces a change in the values of the client variables that has to be reported to the external application.

Figure 2.10b shows a chronograph of the asynchronous link in action. Note that phases 1 and 2 are only executed when the end user interacts with the client application. This fact speeds up the remote simulation reducing the idle time in the asynchronous remote link since, unlike the synchronous version, the external application does not waste time waiting for any information from the client application.

The implementation of the asynchronous remote link leads to implement an extended version of the synchronous remote implementation of the external application.

The extension of the implementation requires among other things a suitable codification of the `synchronize` method, which needs now to update the external application with all the values of linked client's variables. The `synchronize` method must be used by the author in the asynchronous link to inform the external application that a user interaction has taken place. The method will then accomplish phases 1 and 2 of Table 2.1.

To perform the remote operation (using either synchronous or asynchronous links) two new methods can be used by authors when designing the simulation. These methods are listed in Listing 2.9.

```
1 /**
2  * Sets the package size used to group valus of the external variables
3  */
4 public void setPackageSize(int size);
5
6 /**
```

```
7  * Empty the buffer
8  */
9 public void flush();
```

Listing 2.9: New methods required for remote links.

Grouping data in packages is a natural way to send information in commuted networks. The new `setPackageSize` method can be used by authors to increase the amount of information that the external application sends to the client application in one package of data.

Using packages of suitable size can reduce network delays in some cases, although it can also increase the computational time required to show this information while viewing the simulation. This action can improve the Quality of Service (QoS) of the network application. Determining the optimal size is left to the author, since it depends on the network delays observed or expected, the protocol used for network transport, the computational time of the simulation, and other parameters. More sophisticated way to improve the QoS can be found in (Salzmann et al. 2005, Perritaz et al. 2009).

Figure 2.11 shows a package of data for the bouncing ball simulation. In this simulation, three external variables are being sent to the client application: the time, and the height and speed of the ball at that time. The size of the package indicates how many values of these three variables are sent in one package.

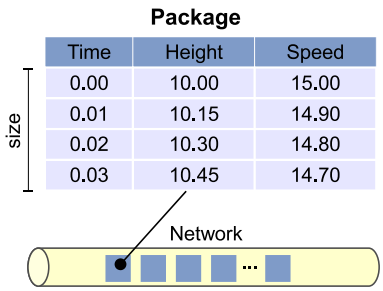


Figure 2.11: A package of data of size 4 is sent from the server side to the client side. The package then transports the first four registers of the bouncing ball simulation.

The sending of packages is controlled automatically by the operating system, according to the transport network protocol implementation. However, the sending can be forced by calling the `flush` method, which allows authors to send packages immediately, without predefined delays.

2.4.4 Synchronous or asynchronous link?

The selection of either the synchronous or asynchronous link for a given remote link should be based on the particular situation expected for the end user. If the remote simulation will be executed through a LAN, the best option is probably to use the synchronous link. This option is the simplest, since it requires only minimal changes of the author on the virtual lab's code to transform the local version to a remote version of the engineering simulation. For execution of the remote simulation through a WAN, authors should favour the asynchronous version to help reduce the network traffic, and hence delays. This choice will however require more changes in the code of the engineering simulation to transform the local version of the simulation into a remote one.

2.5 Advantages of the interoperate approach

This chapter has been on introduction of the interoperate approach to add human interfaces to engineering software, the communication protocol defined by it, and an implementation in Java that supports it.

The main advantage of the approach is that it separates the development of interactive simulations into two qualitatively different parts: the model of the engineering simulation and its human interface.

This scheme helps instructors design and implement the interactive simulations in two separated activities. On the one hand, the creation of the model of the simulation can be done using a specialized engineering software. On the other hand, the design of the user interface to display the response of the system and to capture user interaction can be done using graphics-enabled programming languages or authoring tools. Notice that this is not the common situation when instructors design simulations for pedagogical purposes, because they are normally experts mainly in the building of the engineering simulation, but not in the programming of the interactive user interface.

The approach can also add robustness to the instructor's application, because the author can have a list of replacements, to an initial engineering software, that can accept the role of external applications. Thus, external applications that can be used as alternative, in case the connection with a primary engineering software fails. Using an suitable use of this list, the author's application should seek the first available external

protocol and redirect all future commands to it. This nice feature is implementing in the use of the interoperate approach from Easy Java Simulations in Chapter 4.

A third important advantage of the approach is the possibility of reusing a given user interface to manipulate different engineering simulations which model essentially the same system. This feature of the approach helps reduce the development time of interactive simulations. For instance, an existing human interface that students are familiar with, and can use comfortably, can be used again to control a new simulation with a totally different engineering software. Figure 2.12 visualizes this idea. In it, the same user interface can be selected to control one of two distinct engineering simulations. Note that one of these engineering simulations is located on a remote computer.

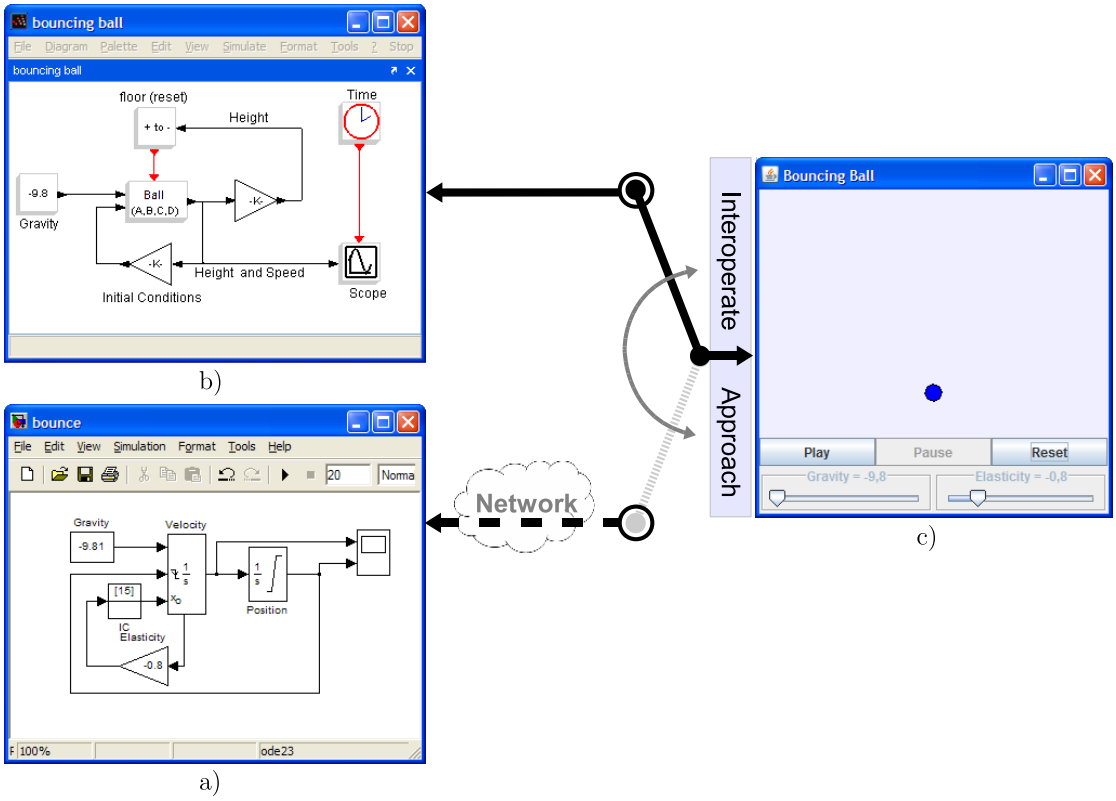


Figure 2.12: The Interoperate Approach allows to use the same simulation GUI with different engineering simulations. The figures correspond to: a) The Simulink model of the bouncing ball, b) The Scicos model of the bouncing ball, c) The human interface of the interactive simulation.

2.6 Conclusions

This chapter has focused on the definition of a communication protocol API for engineering software, and on the implementation of high-level and low-level protocols that it uses to support the interoperate approach.

The final aim of the interoperate approach is to highlight the design and implementation of the human interfaces as totally different activities in the creation of engineering simulations. For this reason, new methods and tools are required to produce virtual labs with high graphical level to represent the response of the system and especially oriented towards capturing the user interaction.

The next chapter describes the technical details of the implementation of the low-level protocol for different standard engineering programs. Chapter 4 describes how the authoring tool Easy Java Simulations has been modified to naturally support the interoperate approach, thus helping instructors in the process of building educational engineering simulation with interactive human interfaces.

Chapter 3

Implementing the Interoperate Approach

Nowadays, most engineering software tools provide an interface for their external connection from other programming languages. This connection offers the opportunity to control these software tools from practically any kind of application.

This external interface can profit from the interoperate approach to add human interfaces to the simulations created by using a particular engineering software.

The communication protocol for local connections defined in Chapter 2 will be implemented in this chapter for three well-known engineering software tools: MATLAB/Simulink, Scilab, and Sysquake. An extension for a remote link with MATLAB/Simulink will also be described, given the popularity of this engineering software. The low-level protocols with engineering software is a generalization of previous experiences with MATLAB described in (Sánchez 2001, Sánchez et al. 2002, 2004, 2005, Buccieri et al. 2005, Hernandez et al. 2008, Guzmán et al. 2005, Martín et al. 2004). The high-level protocols, for MATLAB and others engineering software, described in this chapter are totally novel. The final result of these implementations are software components freely available to the engineering education community.

3.1 Interfacing MATLAB with standard languages

MATLAB is a high-performance language for technical computing (The MathWorks 2010, The MathWorks 2009*b*). Figure 3.1 shows the typical elements that this engineering software offer to users.

The tool provides a system whose basic data element is an array that does not require dimensioning. This allows the user to solve many technical computing problems,

especially those with matrix and vector formulae, in a fraction of the time it would take to write a program in a scalar non interactive language such as C or Fortran. Typical uses include:

- Math and computation.
- Algorithm development.
- Data acquisition.
- Modelling, simulation, and prototyping.
- Data analysis, exploration, and visualization.
- Scientific and engineering graphics.
- Application development, including graphical user interface building.

The name MATLAB stands for matrix laboratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB engines incorporate the LAPACK and BLAS libraries, embedding the state of the art in software for matrix computation. MATLAB has evolved over a period of years until becoming the *de facto* standard in many fields of engineering such as control engineering. In university environments, MATLAB is the standard

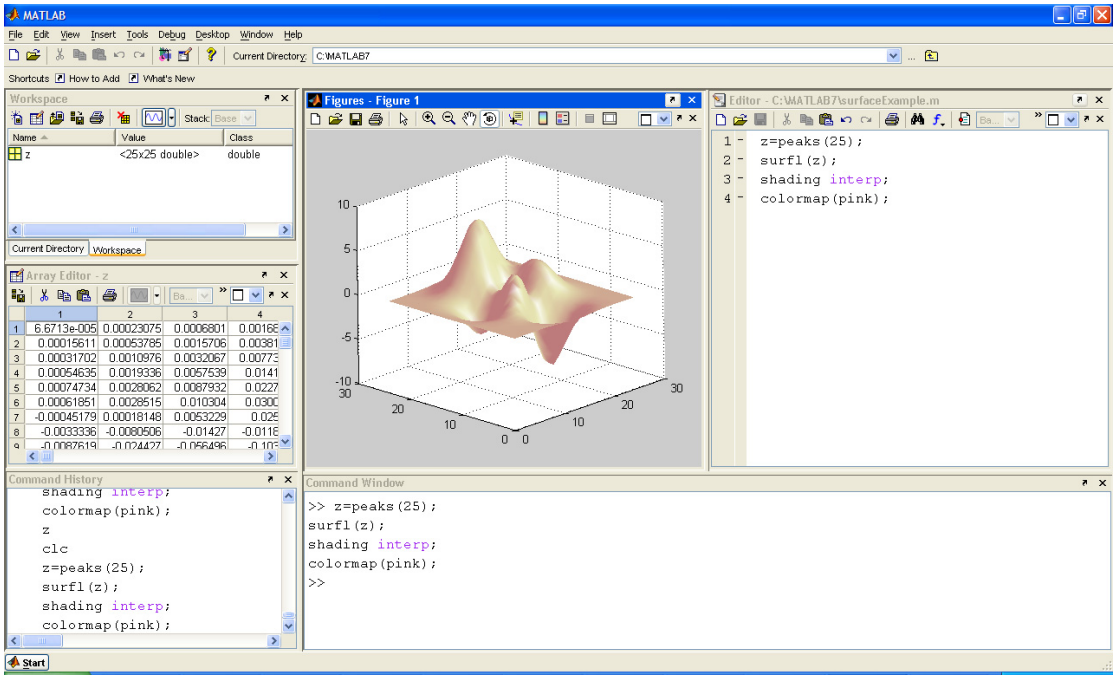


Figure 3.1: The MATLAB environment.

instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of add-on application-specific solutions called toolboxes. These toolboxes allow users to learn and apply specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

Because of this widespread use, MATLAB was selected as the main engineering software tool to apply the interoperate approach to. Next sections will show how the local and remote connections were implemented to support the communication protocol. There is also a special implementation for the Simulink toolbox, because of the many existing Simulink simulations which have been developed for years by the control engineering community.

The results of the implementation process are two Java software components called **JIM Client** and **JIM server**, which are freely available for the education engineering community (Department of Computer Science and Automatic Control, UNED 2010*b*,*a*). Both software components allow engineering teachers to use them to manipulate MATLAB and Simulink directly from Java programs.

This section focuses on the implementation of the communication protocol for a local connection with MATLAB. The analysis starts with a detailed discussion about how this local link was implemented, showing the main elements required to support the local connection with MATLAB. The sections ends describing a Java library that implements the communication protocol, which can be used by instructors to build interactive simulations with MATLAB.

3.1.1 Calling MATLAB from C

In order to implement a local link of the communication protocol with MATLAB, it will first be described how MATLAB is controlled from an external program written in the C language. After this, a short description about Java Native Interface (JNI) is given. This framework allows Java programs to call routines or libraries written in other

languages like C. Using the JNI, the Java library JMatLink manipulates MATLAB in a way similar to what the low-level protocol requires. Finally, a Java implementation of the communication protocol which interfaces with MATLAB using the Java library JMatLink will be presented.

The Engine Library

The MATLAB engine library is a set of routines that allows programmers to call MATLAB from custom software, thereby employing MATLAB as a computation engine. MATLAB engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes, on UNIX, and through a Component Object Model (COM) interface, on Windows. There is a library of functions provided with MATLAB that allows programmers to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB (The MathWorks 2009*a*).

Table 3.1 shows the routines of the engine library for C programs in order to control the MATLAB computation engine. The name of these methods all begin with the three-letter prefix *eng*.

A typical example using the MATLAB engine is to call a mathematical routine, for instance, to invert an array or to compute a FFT in MATLAB, and then to get back the result of one of these operations to a front end (GUI), which has been programmed in C. See a simple example in Listing 3.1. In this C program, MATLAB is opened (by `engOpen`) to evaluate in MATLAB the square function of an array of values stored in the variable `X`. This variable and its values are defined in the MATLAB workspace by the method `engPutVariable`. The routine `engEvalString` performs the evaluation of the square function described by the String ``` ``Y = X.^2;'' ```. The results of the computation are obtained by the C program using `engGetVariable`. The program ends printing the array values and the results. Note that some auxiliary routines of `engine.h`, such as `mxCreateDoubleMatrix` or `mxGetPr`, are also used for a suitable treatment of the C and MATLAB variables.

Besides these C functions, the engine library has routines to control the MATLAB engine from Fortran programs or even from MEX-Files[1]. However, in this work, only C

[1]MEX stands for MATLAB Executable. MEX-files are dynamically linked subroutines produced from C, C++, or Fortran source code that, when compiled, can be run from within MATLAB in the same way as MATLAB M-files or built-in functions.

Table 3.1: C Engine Routines.

Function	Description
Engine *engOpen (const char *startcmd)	Start up MATLAB engine. A pointer to an engine handle is returned. On Windows, the startcmd string must be NULL.
int engClose (Engine *ep)	Shut down MATLAB engine. The input parameters is the engine pointer.
mxArray *engGetVariable (Engine *ep, const char *varname)	Get a MATLAB array from the MATLAB engine. The input parameters are the engine pointer and the name of mxArray to get from MATLAB respectively.
int engPutVariable (Engine *ep, const char *varname, const mxArray *arrayptr)	Send a MATLAB array to the MATLAB engine. Input parameters are the engine pointer, the name given to the mxArray in the engine's workspace and the mxArray pointer.
int engEvalString (Engine *ep, const char *string)	Execute a MATLAB command. Input parameters are the engine pointer and the string to execute.
int engOutputBuffer (Engine *ep, char *p, int n)	Create a buffer to store MATLAB text output. Input parameters are the engine pointer, the pointer to character buffer and the length of the buffer.
Engine *engOpenSingleUse (const char *startcmd, void *dcom, int *retstatus)	Start a MATLAB engine session for single, non shared use. On Windows, the startcmd string must be NULL. input parameter dcom is reserved for future use, and it must be NULL. The last input parameter returns the status of the operation.
int engGetVisible (Engine *ep, bool *value)	Determine visibility of MATLAB engine session. The input parameters are the engine pointer and a pointer to value returned from engGetVisible.
int engSetVisible (Engine *ep, bool value)	Show or hide MATLAB engine session. Input parameters are the engine pointer and the value to set the visible property to (1 means visible and 0 invisible).

routines are taken into account since these functions can be directly manipulated from Java programs as shown in the next subsection.

3.1.2 Calling C routines from Java

The Java Native Interface (JNI) enables the integration of code written in the Java programming language with code written in other languages such as C and C++. JNI allows programmers to take full advantage of the Java platform without having to abandon their investment in legacy code (Liang 1999).

The very simple, *hello world* program shown in Listing 3.2 illustrates briefly how JNI works in Windows systems. A native method declaration must contain the *native*

```
#include "engine.h"
...
int main() {
 Engine *ep; int i; mxArray *X = NULL, *Y = NULL; double *x, *y;

 // Create the variable X
 X = mxCreateDoubleMatrix(1, 10, mxREAL);
 x = (double *)mxGetPr(X); for(i=0;i<10;i++) x[i] = (double)i;

 // Open MATLAB
 ep = engOpen("\0");

 // Place variable X into MATLAB workspace
 engPutVariable(ep, "X", X);

 // Evaluate Y=X*X
 engEvalString(ep, "Y=X.^2;");

 // Get result of the function evaluation
 Y = engGetVariable(ep,"Y")
 y = (double *)mxGetPr(Y);

 //Print X and Y
 printf("\n"); for(i=0;i<10;i++) printf("x[%d]=%f\n",i,x[i]);
 printf("\n"); for(i=0;i<10;i++) printf("y[%d]=%f\n",i,y[i]);

 //Free memory
 mxDestroyArray(X); mxDestroyArray(Y);

 //Close MATLAB
 engClose(ep);
}
...
```

Listing 3.1: An example of using MATLAB engine from C.

modifier. This declaration means that the native method is implemented in another language. That is the reason why the `print()` method has no implementation in the class itself. In this case such implementation is given by the C file of Listing 3.3.

However, before using the `print()` method, the C implementation has to be compiled as a native library named `HelloWorld.dll`. This library is loaded into the system memory with the Java method `System.loadLibrary`. Once the library is loaded into memory, the native `print()` method can be invoked as a standard Java method.

```
class HelloWorld {
  //Native method
  private native void print();

  public static void main(String[] args) {
   //Load HelloWorld.dll in system memory
   System.loadLibrary("HelloWorld");

   //Call to the native method print
   new HelloWorld().print();
  }
}
```

Listing 3.2: A Java program declaring and using JNI.

To implement the native method, the programmer needs first to compile the Java

class HelloWorld of Listing 3.2 by using the command **javac HelloWorld.java**. A header file, named `HelloWorld.h`, must also be created to support the implementation of the native method. The header file is generated by using the Java command **javah -jni HelloWorld**. This file provides the programmer with the following C declaration (i.e. the *signature*) of the `print()` method:

```
JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj)
```

This declaration is then used in the C file that implements the native method as Listing 3.3 shows. A more detailed description about implementing the Java Native Interface can be found in (Liang 1999).

```
1 #include <jni.h>
2 #include <stdio.h>
3 #include "HelloWorld.h"
4 JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj)
5 {
6   printf("Hello World!\n");
7   return;
8 }
```

Listing 3.3: The implementation of the JNI print method.

3.1.3 Calling MATLAB from Java

With the previous discussion about the implementation of JNI method in mind, the next step is to code a Java library to call the native methods of the MATLAB engine library. Fortunately, the existing **JMatLink** (Müller 2010) library does exactly this. This Java library is an open source project available at:

`http://jmatlink.sourceforge.net/index.php`

JMatLink works exactly as explained above to support the native methods that call the routines of the engine library. Check for instance the implementation in Listing 3.4 of the native `engEvalStringNATIVE` method. This native method is declared in the `CoreJMatLink` class and is used to call the engine routine `engEvalString`. All the native methods are implemented in the `jmatlink.dll` native library.

```
1  ...
2  /**
3   * int engEvalStringNATIVE( long epI, String evalS )
4   * @param  epI
5   * @param  evalS
6   * @return
7   */
8  JNIEXPORT jint JNICALL Java_jmatlink_CoreJMatLink_engEvalStringNATIVE
9                  (JNIEnv *env, jobject obj, jlong enginePtr, jstring evalS_JNI)
10 {
```

```
11   int  retValI = 0;
12   const char *evalS = (*env)->GetStringUTFChars(env, evalS_JNI, 0);
13
14   // evaluate expression in MATLAB
15   retValI = engEvalString((Engine *)enginePtr, evalS);
16
17   if (retValI != 0)
18         printf("engEvalStringNATIVE: return value !=0, some error\n");
19
20   (*env)->ReleaseStringUTFChars(env, evalS_JNI, evalS); // free memory
21
22   return retValI;
23  }
24  ...
```

Listing 3.4: Part of jmatlink.dll to implement the native method engEvalStringNATIVE.

The package consists of three following classes:

`JMatLink`, `CoreJMatLink`, and `JMatLinkException`.

The first class, JMatLink, provides all the public Java methods that other Java programs can use to connect to MATLAB. The second class, CoreJMatLink, as its name indicates, is a utility class that represents the core of the JMatLink package. CoreJMatlink extends the standard Java class Thread to keep continuous information of the various MATLAB engines. In this class, all the native methods (such as engEvalStringNATIVE commented above) are declared. The third class, JMatLinkException, is used by JMatLink to treat and throw exceptions of the native link. So, both CoreJMatlink and JMatLinkException are used internally by JMatLink and should not be called directly by programmers.

JMatLink offers Java methods similar to the C routines of the engine library of MATLAB. Table 3.2 shows the main methods of JMatLink.

Listing 3.5 shows an example of use of JMatLink. The purpose of this simple Java program is to compute the inverse of a matrix. The matrix x to be inverted is defined in line 15. Then, in order to connect to MATLAB, a new instance of JMatLink, called `matlab`, is created. This instance of JMatLink offers the programmer all the methods provided by the JMatLink library. A MATLAB console is opened by using `engOpenSingleUse`. This function returns an identifier for the MATLAB session. Note that other consoles can also be opened by repeatedly calling the method `engOpenSingleUse`. With the primitive `engPutArray`, the matrix x is sent to the MATLAB workspace. The inverse of the matrix x is computed with the execution of the method `engEvalString(ep,"Y=inv(X)")`. The inverted matrix is then saved in the variable `Y`, which is recovered from the MATLAB workspace with `engGetArray`. Finally, both matrices, x and y, are printed.

Table 3.2: Main methods of the library JMatlink.

Method	Description
void engClose(long epI)	Close a specified connection to an instance of MATLAB.
void engEvalString(long epI, java.lang.String evalS)	Evaluate an expression in a specified workspace.
double[][] engGetArray(long epI, java.lang.String arrayS)	Get an array from a specified instance/-workspace of MATLAB.
java.lang.String[] engGetCharArray(long epI, java.lang.String arrayS)	Get an 'char' array (string) from MATLAB's workspace.
java.lang.String engGetOutputBuffer(long epI)	Return the ouputs of previous commands in MATLAB's workspace.
boolean engGetVisible(long epI)	Return the visibility status of the MATLAB window.
long engOpenSingleUse()	Open engine for single use and returns the identifier of the MATLAB session.
void engPutArray(long epI, java.lang.String arrayS, double valueD)	Put an array into a specified instance/-workspace of MATLAB.
void engPutArray(long epI, java.lang.String arrayS, double[] valuesD)	Put an array (1 dimensional) into a specified instance/workspace of MATLAB.
void engPutArray(long epI, java.lang.String arrayS, double[][] valuesDD)	Put an array (2 dimensional) into a specified instance/workspace of MATLAB.
void engSetVisible(long epI, boolean visB)	Set the visibility of the MATLAB window.

```
1  import jmatlink.*;
2
3  public class testjmat {
4
5    //Declare variables
6    double[][] x;
7    double[][] y;
8
9    public static void main (String[] args) {
10         new testjmat();
11   }
12
13   public testjmat() {
14     // Set the matrix x
15     x = new double[][]{{1,-2,4},{3,1,0},{-1,1,2}};
16
17     // Create instance of JMatLink
18     JMatLink matlab = new JMatLink();
19
20     // Open and start MATLAB
21     long ep = matlab.engOpenSingleUse();
22
23     // Put x into MATLAB
24     matlab.engPutArray(ep, "X", x);
25
26     // Compute in MATLAB the inverse of x
27     matlab.engEvalString(ep,"Y=inv(X)");
28
29     //Get the inverse of x
30     y = matlab.engGetArray(ep,"Y");
31
32     //Print x and y
33     for(int i=0;i<3;i++)
34       for(int j=0;j<3;j++) System.out.println("x("+i+","+j+")="+x[i][j]);
35
36     for(int i=0;i<3;i++)
```

```
37      for(int j=0;j<3;j++) System.out.println("y("+i+","+j+")="+y[i][j]);
38
39      // Close MATLAB
40      matlab.engClose(ep);
41    }
42  }
```

Listing 3.5: Computing the inverse of matrix using JMatLink.

JMatLink offers a very good connection with MATLAB, allowing programmers full control of the engine library using Java. However, the methods provided by JMatLink lack the functionality required by the communication protocol of the Interoperate Approach. This disadvantage is covered in the following section.

3.2 An implementation of `ExternalApp` for MATLAB

Observing the methods of JMatLink described in Table 3.2, it can be concluded that much of the functionality required by the low level protocol is covered. Actions like setting and getting values from MATLAB are done directly with the JMatLink methods `engPutArray` and `engGetArray`, respectively. Evaluating commands in MATLAB is also provided by JMatLink through the `engEvalString` method. Even more, the performance of the JMatLink methods to open (`engOpenSingleUse`) and close (`engClose`) MATLAB, is quite similar to that required by the methods `connect` and `disconnect` of the high level protocol. However, other actions like linking external and client variables, or stepping (i.e, executing a predefined command and updating variables), are clearly not supported by JMatLink.

Therefore a new layer of functionality has to be added to JMatLink so that it implements the capability of interfacing with MATLAB required by the communication protocol. Hence, from a software point of view, the Java library required to support an implementation of the `ExternalApp` interface should be composed like Figure 3.2 shows.

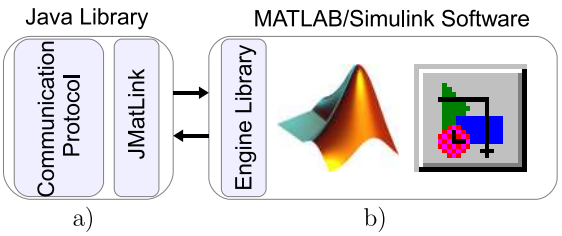


Figure 3.2: The local connection with MATLAB/Simulink.

3.2.1 The MatlabExternalApp Java class

In order to support of the interoperate approach with MATLAB, a Java class called MatlabExternalApp was coded following the requirements of the interface ExternalApp. As it was said before, the implementation of the low level protocol using JMatLink is quite direct. For example, Listing 3.6 shows the Java code that supports the functionality needed by the actions to evaluate a command in MATLAB (eval), to set the value of an array in MATLAB (setValue), and to get the value of an array from MATLAB (getDoubleArray). Note that the MatlabExternalApp class implements the Java interface ExternalApp defined previously in Chapter 2.

```
1  ...
2  public class MatlabExternalApp implements ExternalApp{
3    ...
4    public void eval (String command) {
5      if (matlab==null) return;
6      matlab.engEvalString (id,command);
7    }
8
9    public void setValue (String name, double[] value) {
10     if (matlab==null) return;
11     matlab.engPutArray (id,name,value);
12   }
13
14   public double[] getDoubleArray (String variable) {
15     if (matlab==null) return null;
16     double[][] returnValue = matlab.engGetArray(id,variable);
17     return returnValue[0];
18   }
19   ...
20 }
```

Listing 3.6: MatlabExternalApp class: Implementation of methods eval, setValue, and getDoubleArray.

To support the high-level protocol, the implementation of the methods of the interface ExternalApp is more elaborate than before. For instance, Listing 3.7 shows the methods connect and disconnect that start and finish the connection with MATLAB.

```
1  public boolean connect (){
2    if (matlab==null){
3      matlab = new JMatLink();
4      id = matlab.engOpenSingleUse();
5      return true;
6    }
7    ...
8  }
9
10 public void disconnect (){
11   if (matlab!=null){
12     matlab.engClose(id);
13     matlab=null;
14     ...
15   }
16 }
```

Listing 3.7: MatlabExternalApp class: Implementation of methods connect and disconnect.

Another, very important method required by the high level protocol is `setClient`, which provides the external application with a pointer to the client application. This pointer is used to link the client and external variables. The implementation of `setClient` in the `MatlabExternalApp` class is shown in Listing 3.8. The method just saves two necessary variables, the Object client (`varContextObject`) and an array of the Field (i.e., the public variables and methods) of the client (`varContextFields`). Both variables will be used to perform the linking of external and client variables.

```
1  import java.lang.reflect.*;
2  ....
3  public class MatlabExternalApp implements ExternalApp{
4    ...
5    // The client of the external application
6    protected Object varContextObject=null;
7    //An array of fields of the client
8    protected Field[] varContextFields=null;
9    ....
10
11   public boolean setClient (Object client) {
12     if (client==null) return (false);
13     varContextObject=client;
14     varContextFields=client.getClass().getFields();
15     return (true);
16   }
17   ...
18 }
```

Listing 3.8: `MatlabExternalApp` class: Implementation of methods setClient.

As it was also mentioned in Chapter 2, the linking of variables is done using the Java *Reflection* API. This method uses the ability of the reflection feature to examine or modify the runtime behaviour of applications running in the Java virtual machine. Thus, by using reflection, the values of client public variables can be set to the values obtained from the external variables.

Listing 3.9 shows the implementation of the `linkVariables` method, which links client and external variables. Note that this method looks for the variable named `cvar` in the array of client variables `varContextFields`. If the client variable `cvar` exists, then the method obtains the type of the variable. The type of the client variable is obtained by calling the reflection API:

```
varContextFields[i].getType().getName()
```

The call to `getName()` returns a String which defines the type of the client variable. The string ‘‘`[[D`’’, for instance, represents a matrix of doubles.

Take into account that not all Java types are supported. Only the **double**, **double**[], **double**[][], and **String** types can be linked with external variables, since these types

suffice to work with most MATLAB applications. Once the type of the variable `cvar` is found, the type itself and the index of the variable `cvar` in the array `varContextFields` are saved (in `linkIndex` and `linkType` respectively) for future use by the `setValues` and `getValues` methods. Finally, the name of both client and external variables (`cvar` and `evar`) are saved in the `linkVector` Vector.

```
1  ...
2  public boolean linkVariables(String cvar, String evar){
3    if (varContextObject==null) return (false);
4    int type;
5    //Search if the cvar exists
6    for (int i=0; i < varContextFields.length; i++) {
7     if (cvar.equals((varContextFields[i]).getName())) {
8      //Detect type
9      if (varContextFields[i].getType().getName().equals("double"))
10           type=DOUBLE;
11     else if (varContextFields[i].getType().getName().equals("[D"))
12       type=ARRAYDOUBLE;
13     else if (varContextFields[i].getType().getName().equals("[[D"))
14       type=ARRAYDOUBLE2D;
15     else if
           (varContextFields[i].getType().getName().equals("java.lang.String"))
16       type=STRING;
17     else return (false);
18     if (linkVector==null) {
19       linkVector=new java.util.Vector();
20       linkIndex= new int[1];
21       linkIndex[0]=i;
22       linkType = new int[1];
23       linkType[0]=type;
24     }else{
25       int[] _linkIndex=new int[linkIndex.length+1];
26       System.arraycopy(linkIndex,0,_linkIndex,0,linkIndex.length);
27       _linkIndex[linkIndex.length]=i;
28       linkIndex=_linkIndex;
29       int[] _linkType=new int[linkType.length+1];
30       System.arraycopy(linkType,0,_linkType,0,linkType.length);
31       _linkType[linkType.length]=type;
32       linkType=_linkType;
33     }
34     linkVector.addElement(cvar+";"+evar);
35     return (true);
36    }
37   }
38   return(false);
39 }
40 ...
```

Listing 3.9: `MatlabExternalApp` class: Implementation of the linkVariables method.

The linked variables are updated every time the method `step` is called. The implementation of this function is shown in Listing 3.10. The execution of each step implies a call to the methods `setValues`, `eval`, and `getValues`. The method `eval` is executed as many times as the parameter `dt` indicates. The implementation of the methods `setValues` and `getValues` is shown in Listing 3.11.

When the method `setValues` is called by `step`, all the information saved by the `linkVariables` about the linked variables is used to update the external variables. This

```
...
public void step (double dt) {
  //Set all external variables
  setValues();

  //Evaluate the command
  int steps=(int)dt;
  for (int i=0;i<steps;i++)
    eval(command);

  //Set all client variables
  getValues();
}
...
```

Listing 3.10: `MatlabExternalApp` class: Implementation of the step method.

modification of the values of the external variables is done according to the type of the linked client variable. Thus, if the type of a client variable is an array of doubles, then its value is used to modify the corresponding external variable by calling the method `setValue` of Listing 3.6. Again, the reflection API is used, in this case to get the value of the client variable by calling:

```
(double[])varContextField.get(varContextObject)
```

Note that the values obtained by the reflection method `get` have to be converted to the correct type using the corresponding type casting (e.g. `double[]`). Note also that in the case of the `double` type, the method of the reflection API used is `getDouble`.

```
...
public void setValues(){
  int k=0;
  String cvar,evar;
  for (int i=0; i<linkVector.size(); i++){
    cvar= linkVector.elementAt(i);
    evar=cvar.substring(cvar.indexOf(";")+1,cvar.length());
    try {
      Field varContextField = varContextFields[linkIndex[k]];
      switch (linkType[k++]){
        case DOUBLE:
          setValue(evar,varContextField.getDouble(varContextObject));break;
        case ARRAYDOUBLE:
          setValue(evar,(double[])varContextField.get(varContextObject));break;
        case ARRAYDOUBLE2D:
          setValue(evar,(double[][])varContextField.get(varContextObject));break;
        case STRING:
          setValue(evar,(String)varContextField.get(varContextObject));break;
      }
    } catch (java.lang.IllegalAccessException e) {
      System.out.println("Error Step: setting a value " + e);
    }
  }
}

public void getValues(){
  int k=0;
  String cvar,evar;
  for (int i=0; i<linkVector.size(); i++){
    cvar= linkVector.elementAt(i);
```

```
31        evar=cvar.substring(cvar.indexOf(";")+1,cvar.length());
32        try {
33          Field varContextField= varContextFields[linkIndex[k]];
34          switch (linkType[k++]){
35            case DOUBLE:
36              varContextField.setDouble(varContextObject,getDouble(evar));break;
37            case ARRAYDOUBLE:
38              varContextField.set(varContextObject,getDoubleArray(evar));break;
39            case ARRAYDOUBLE2D:
40              varContextField.set(varContextObject,getDoubleArray2D(evar));break;
41            case STRING:
42              varContextField.set(varContextObject,getString(evar));break;
43          }
44        } catch (java.lang.IllegalAccessException e) {
45          System.out.println("Error Step: getting a value " + e);
46        }
47      }
48    }
49    ...
```

Listing 3.11: `MatlabExternalApp` class: Implementation of setValues and getValues methods.

Regarding the method `getValues`, the process is obviously inverse to `setValues`. The values of the external variables are obtained with the corresponding methods to get values from MATLAB (e.g. getDoubleArray of Listing 3.6). These values are then used to modify the respective client variables by calling the reflection API as follows:

```
varContextField.set(varContextObject,getDoubleArray(evar))
```

Note that, except for `doubles`, the `set` method of the reflection API is used to modify a client variable of any type. In the case of `doubles` the correct method of the reflection API is `setDouble`.

The rest of the Java class `MatlabExternalApp` is not very different from what has been described so far. The next subsection will show how the class `MatlabExternalApp` can be used to build Java applications that use the communication protocol with MATLAB.

3.2.2 Using the class `MatlabExternalApp` from a Java program

Once the `MatlabExternalApp` Java class has been described, it can be used to create interactive Java simulations with MATLAB.

A first example comes from revisiting the Listings 2.5 and 2.8 of Chapter 2. Those listings can be easily transformed to be used with the `MatlabExternalApp` class, just by replacing the line:

```
ExternalApp externalApp = new MyExternalApp();
```

With the following one:

```
ExternalApp externalApp = new MatlabExternalApp();
```

Since all methods of the `MatlabExternalApp` class are implemented following the requirements of `ExternalApp`, the execution of this new example produces exactly the same output for the evaluation of the function as before.

A slightly more sophisticated example of the use of the `MatlabExternalApp` class can be achieved if, instead of printing in the console the values of the function, these values are drawn in a plot. To do that, the Java features of the packages **AWT** and **Swing** can be used to generate a graphical user interface that plots the function outputs. Moreover, other nice visual elements such as buttons, a text field, and a slider can be added to the user interface in order to capture user interaction. Such an interactive application is presented in Figure 3.3.

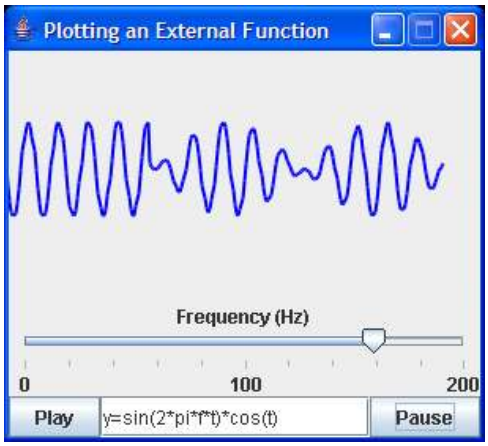


Figure 3.3: An interactive application using MATLAB.

Using this application, users can input directly in the text field the function to be evaluated by MATLAB. For instance, the plot shows the output for the function $y = sin(2 * pi * f * t) * cos(t)$. Another interaction provided by the application is offered by dragging the slider. This visual element controls the value of the variable `f`, which in the case of the function $y = sin(2 * pi * f * t) * cos(t)$, changes the frequency of the sine. Finally, the two buttons, `Play` and `Pause`, allow users to start or pause the execution of the application.

The source code of the application that plots the function is shown in Listing 3.12. The code starts importing the Java packages AWT and Swing required for the visual elements. After that, the client and other auxiliary variables are declared. The graphical interface is implemented in the `evaluatingFunctionPlot` method. Here, the buttons

(Play and Pause), the slider, and the text field are initialized. Note how the user interaction to the text field is added. This action allows users to introduce a new function to be evaluated in MATLAB by using the `setCommand` method. Then, the connection to the `MatlabExternalApp` class is configured. To perform the simulation, a do-while cycle is used. However, on this occasion, instead of printing the output values of the function, these values are drawn as a trace in a plot.

The trace is simply a line between two points (`point1 point2`). The points are obviously the last and the current values of the function output. The object `plot` is an instance of the `PlotPanel` class which is, in turn, an instance of the `Canvas` Java class. The `PlotPanel` overrides the `update` method of the `Canvas` class. The `update` method is called in response to a call to `plot.repaint()`. The plot is first cleared by filling it with the background color, and then completely redrawn by calling the `update` method. Finally, the action performed by `checkBorders` makes sure that the trace is drawn inside the borders of the plot. This is done using adequately the `copyArea` method, which moves to the left the trace drawn before adding a new line. The application ends and disconnects with MATLAB when the user clicks on the close button of the window.

The two applications described are just very simple examples of what can be done by using the `MatlabExternalApp` class. More elaborate simulations will be presented in the next chapter.

3.3 Interfacing Simulink models with Java

A special link with Simulink is now described in detail. Simulink is one of the most famous toolboxes of MATLAB. There is a huge number of simulations built using Simulink models. For this reason, it was decided to develop a dedicated connection with this toolbox in order to facilitate the creation of interactive simulations using Simulink.

Simulink is an environment for multi domain simulation and Model-Based Design for dynamic and embedded systems (see Figure 3.4). It provides an interactive graphical environment and a customizable set of block libraries that let users design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing.

Simulink is integrated with MATLAB, providing immediate access to an extensive range of tools that let users develop algorithms, analyze and visualize simulations, cre-

```
1  //Import packages
2  import javax.swing.*;
3  import java.awt.*;
4  ...
5  public class evaluatingFunctionPlot{
6    //Declare the client and auxiliary variables
7    public double time=0, frequency=1, value=1;
8    ...
9
10   public static void main(String args[]){
11     new evaluatingFunctionPlot();
12   }
13
14   public evaluatingFunctionPlot(){
15     //Create visual elements
16     JFrame frame = new JFrame("Plotting an External Function");
17     plot = new PlotPanel();
18     slider = new JSlider(minFreq,maxFreq,(int)(frequency));
19     functionText = new JTextField("y=sin(2*pi*f*t)*cos(t)");
20     ...
21     //Add interaction to the text field
22     ActionListener alText = new ActionListener() {
23       public void actionPerformed(ActionEvent actionEvent) {
24         externalApp.setCommand(functionText.getText());
25       }
26     };
27     functionText.addActionListener(alText);
28     ...
29     //Create Matlab connection and Link variables
30     externalApp = new MatlabExternalApp();
31     ...
32     //Perform the simulation
33     do{
34       if (!pauseSimulation){
35         externalApp.step(1);
36         point1 = point2;
37         point2 = new Point2D.Double(time,value);
38         line = new Line2D.Double(point1, point2);
39         checkBorders();
40         plot.repaint();
41         time = time+dt;
42       }
43       delay(10);
44     }while(true);
45   }
46   ...
47   public class PlotPanel extends Canvas{
48     public void update (Graphics g) {
49       ...
50       //Draw the trace
51       ((Graphics2D)g).draw(line);
52       //Move the trace
53       if (delta>0) g2D.copyArea(ini,0,fin,sizeY,-delta,0);
54       ...
55     }
56   }
57   ...
58 } //end of evaluatingFunctionPlot
```

Listing 3.12: Computing a Function Using the High Level Protocol.

ate batch processing scripts, customize the modelling environment, and define signal, parameter, and test data (The MathWorks 2009*c*).

Some of the key features of Simulink are its:

- Ability to manage complex designs by segmenting models into hierarchies of design

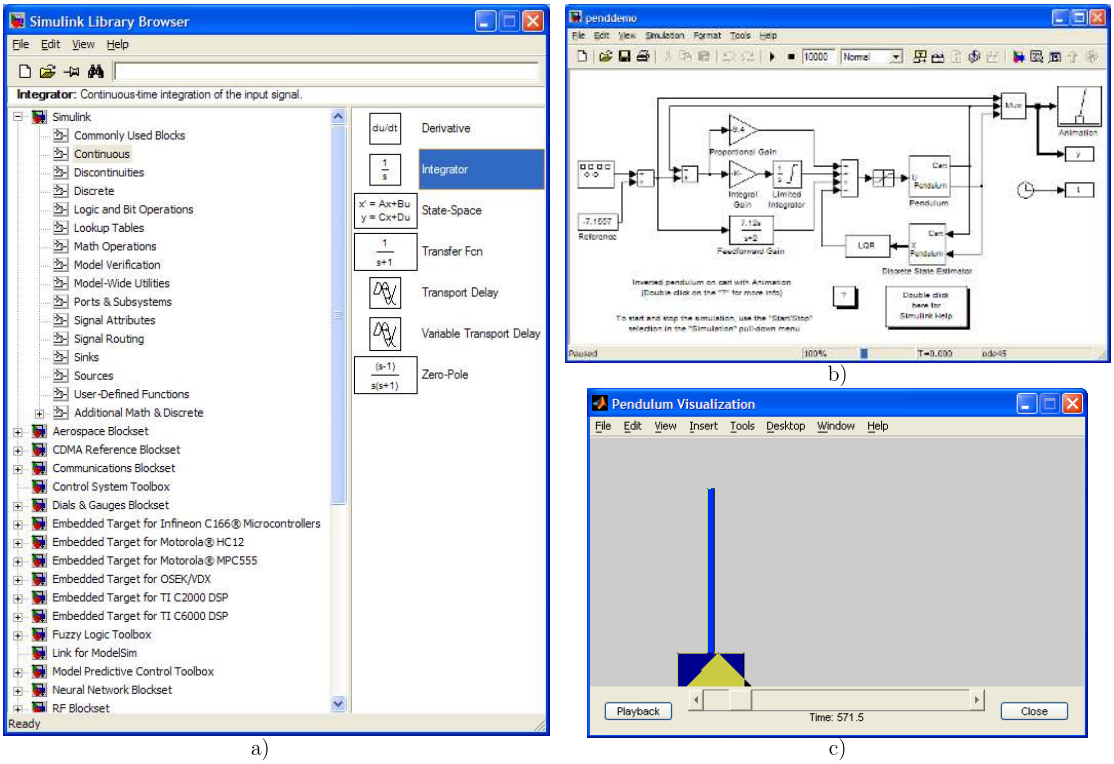


Figure 3.4: Simulink environment. a) The block library of Simulink, b) a Simulink model, and c) an animation of the model.

components.

- Extensive and expandable libraries of predefined blocks.
- Availability of Embedded MATLAB Function blocks for bringing MATLAB algorithms into Simulink and embedded system implementations.
- Model Explorer to navigate, create, configure, and search all signals, parameters, properties, and generated code associated with your model.
- Simulation modes (Normal, Accelerator, and Rapid Accelerator) for running simulations interpretively or at compiled C-code speed using fixed- or variable-step solvers.
- Full access to MATLAB for analyzing and visualizing results, customizing the modelling environment, and defining signal, parameter, and test data.
- Application programming interfaces (APIs) that let users connect with other simulation programs and incorporate hand-written code.
- Interactive graphical editor for assembling and managing intuitive block diagrams.

These features have expanded the use of Simulink over many fields of engineering, and, together with MATLAB, it has become a very popular tool in the academic world, specially for engineering education and research purposes.

For modelling purposes, Simulink provides a graphical user interface for building models in the form of block diagrams, using click-and-drag mouse operations. With this interface, users can draw the models just as they would with pencil and paper (or as most textbooks depict them). This is far ahead other simulation packages that require to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. It is also possible to customize existing blocks or create custom ones.

Models are hierarchical, so users can build models using both top-down and bottom-up approaches. Users can view the system at a high level, then double-click blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After a model is defined, it can be simulated using a choice of integration methods (e.g. *Runge-Kutta*), either from the Simulink menus or by entering commands in the MATLAB Command Window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (e.g., Monte Carlo simulations). Using scopes and other display blocks, users can see the simulation results while the simulation is running. In addition, users can change parameters by using dialogue boxes and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for post processing and visualization.

As it can be noticed, the integration between MATLAB and Simulink is complete. Therefore, this fact can be exploited of in order to build interactive simulations of Simulink models. This will be the object of this section.

3.3.1 Controlling Simulink simulations from MATLAB

Controlling Simulink from MATLAB can be done programmatically by using the API that Simulink provides. The API allows users to open, simulate, pause, stop, or close a Simulink model. It is also possible to add or remove blocks from the Simulink model,

and to modify any block parameter while the simulation is stopped or even while it is running.

Table 3.3: Some common Simulink functions.

Function	Description
open_system	Open existing model or block.
close_system	Close open model or block.
add_block	Add new block.
delete_block	Delete a block.
add_line	Add a line.
delete_line	Remove a line.
set_param	Set parameter values for model or block.
get_param	Get simulation parameter values from model.

Table 3.3 shows some common functions of the Simulink API. The functions `open` and `close` open and close a Simulink model, respectively. The model can be modified by adding or deleting blocks with `add_block` and `delete_block`. Functions `add_line` and `delete_line` allow to connect or disconnect blocks.

The function `set_param` modifies a parameter of a block or model, which can affect strongly the behaviour when simulated. This function is used as follows:

```
set_param('blockpath','parameter','value');
```

The `'blockpath'` string defines the path to the block. The path represents the route to the block inside the model. The `parameter` string indicates the parameters that will be modified, and the `value` string is the new value of the modified parameter.

There are some common parameters for the blocks and models, but obviously different blocks can present an important set of different parameters. To get all the parameters of a block, the function `get_param` can be used as follows:

```
params=get_param('blockpath','objectparameters');
```

The `'objectparameters'` string indicates the information of all parameters of the block or model that are requested.

Using the described functions, a programmatic control of a Simulink simulation can be performed. This will be exemplified by using a Simulink model named **fsmk.mdl** that evaluates a function similar to the example described using the Java class `MatlabExternalApp` in Figure 3.3.

The Simulink version of the evaluating function is shown in Figure 3.5. Here the model (Figure 3.5a) uses simple blocks to evaluate the function: $sin(2\pi ft) \cdot cos(t)$.

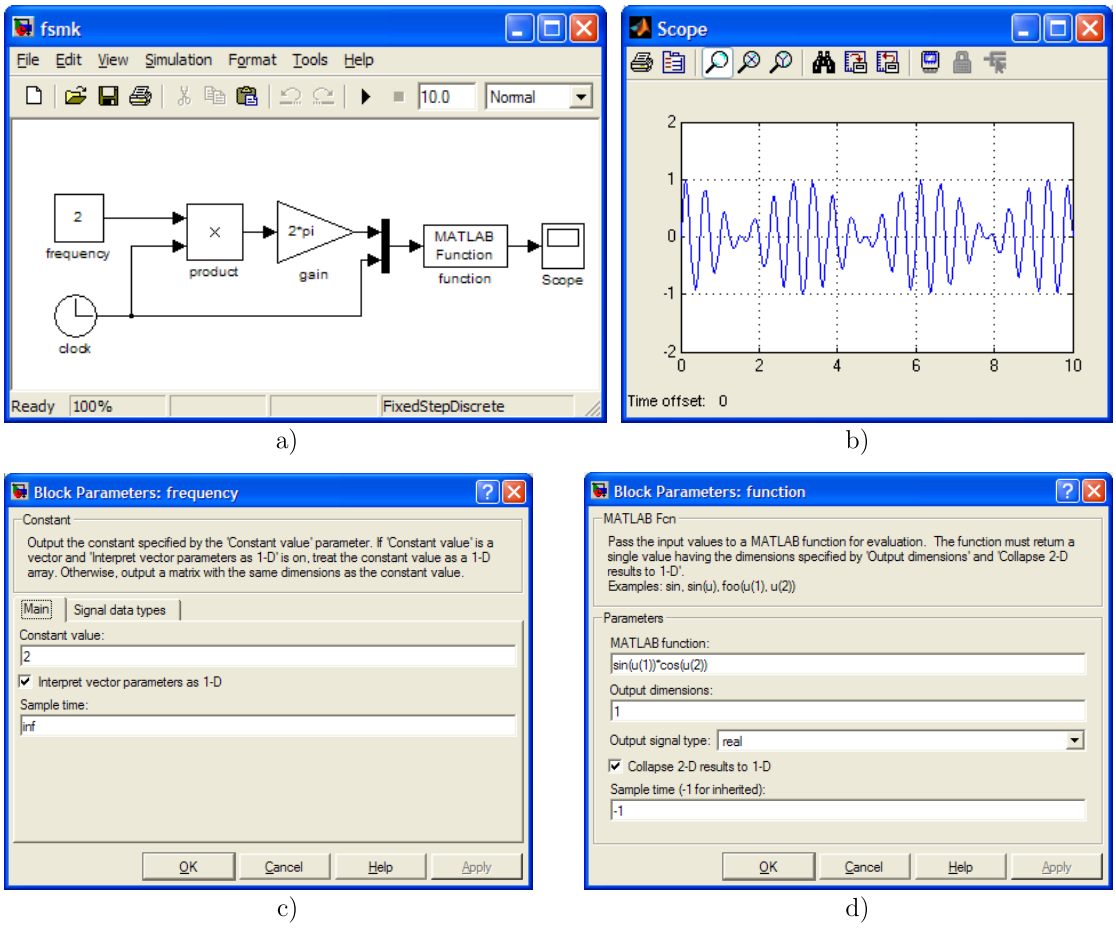


Figure 3.5: Simulink version of evaluating function. a) The model, b) a plot of the function, c) some parameters of the block frequency, d) some parameters of the block function.

The Simulink model needs to be transformed to the one shown in Figure 3.6 by adding standard Simulink blocks. These extra blocks are necessary to manipulate the Simulink model from MATLAB. For instance, the block `FromWS` uses the value of the variable `f` as the frequency of the function. The block `ToWS` modifies the variable `y` with the computed value of the function. The submodel `stepCtrl` allows to add a pause after each integration step, and to write the variable `t` with the value of the simulation time. A more detailed description of this example can be found in Appendix A.

3.3.2 Controlling a modified Simulink model from Java

After modifications of the Simulink model are carried out, the simulation of the Simulink model can be performed from a Java program. Listing 3.13 shows the Java code that simulates the modified Simulink model **fsmkM.mdl** using the `MatlabExternalApp` class.

The code starts declaring the variables and calling the main method of the class `evaluatingFunctionSimulink`. In the method `evaluatingFunctionSimulink()` an

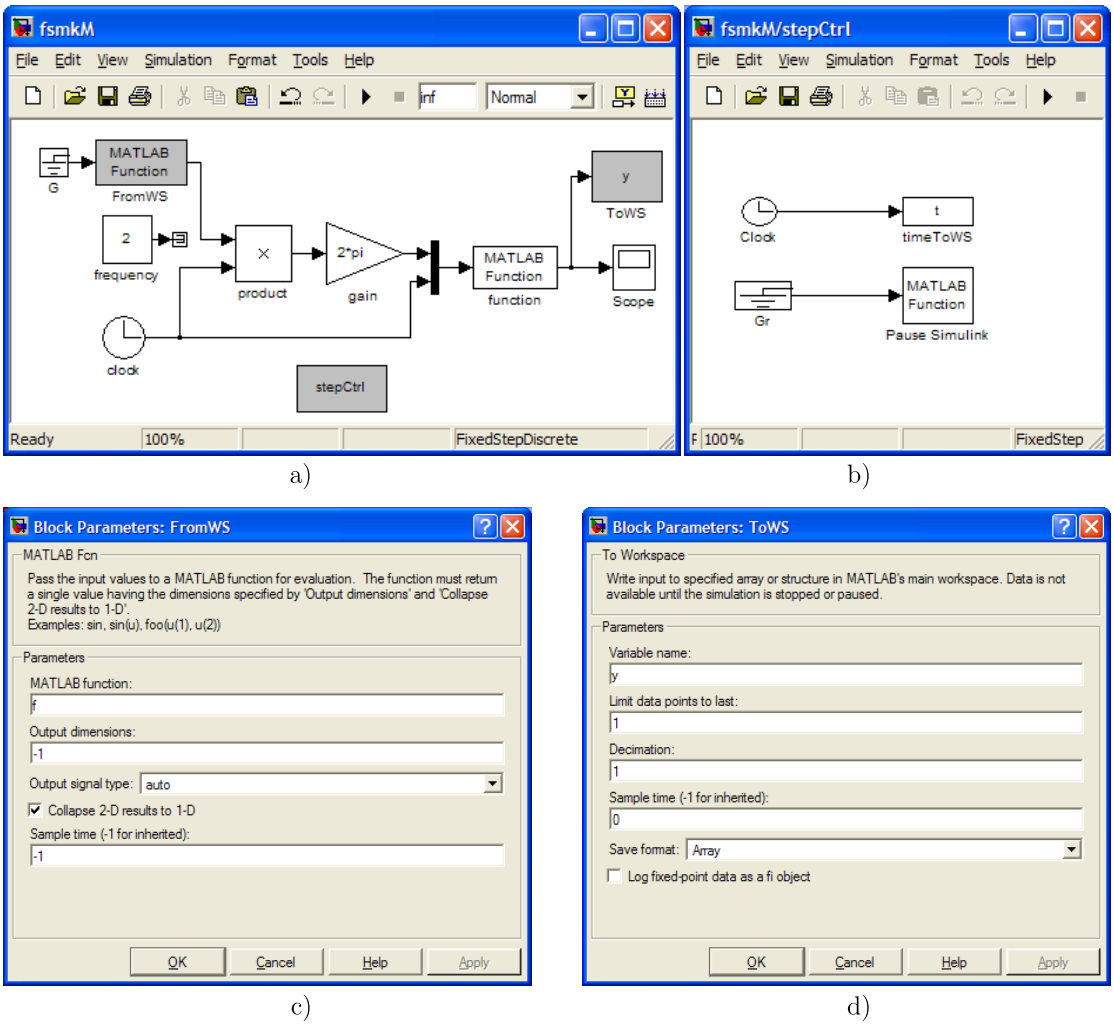


Figure 3.6: A modified version of the Simulink model of the Figure 3.5a. a) The modified model, b) the submodel `stepCtrl`, c) the parameters for the block `FromWS`, and d) the parameters for the block `ToWS`.

instance of `MatlabExternalApp` is first obtained. This object is used to communicate with MATLAB as was explained above. After the connection with MATLAB is started, the Simulink simulation is prepared. To do that, the model is first opened with the MATLAB command `open_system('fsmkM')`. Then, the MATLAB variable `f` is set to the value given by the Java variable `frequency`. Then, the parameter `MATLABfcn` of the block `function` is set to $sin(u(1)) * cos(u(2))$. After that, the simulation is started, with the MATLAB command `set_param('fsmkM','SimulationCommand','start')`. At this moment, the Simulink model is ready to be simulated.

The simulation is run by executing three main action inside of a do-while cycle. The first action steps the Simulink model, which means that the simulation advances one integration step. After that, it is necessary to check if the execution of this integration

step has finished. This checking is done by verifying that the value of the parameter `SimulationStatus` of the model is the `'paused'` string. Take into account that, sooner or later, the Simulink will be paused when the block named `Pause Simulink` is executed. The second action, after this check, is to get the values of the MATLAB variables `t` and `y`, which represent the time and the output of the `function` block. The third action just prints these values to the console.

The do-while cycle is executed until the time is greater than 10. The Simulink model is then stopped and the MATLAB connection finished.

```
1  public class evaluatingFunctionSimulink{
2    //Declare variables
3    public double time=0, frequency=2, value=0;
4    String status;
5
6    public static void main (String[] args) {
7      new evaluatingFunctionSimulink();
8    }
9
10   public evaluatingFunctionSimulink(){
11     //Create a Matlab connection
12     ExternalApp externalApp = new MatlabExternalApp();
13
14     //Start the connection
15     externalApp.connect();
16
17     //Open and prepare simulation
18     externalApp.eval("open_system('fsmkM')");
19     externalApp.setValue("f",frequency);
20     externalApp.eval("set_param('fsmkM/function',
21                      'MATLABfcn','sin(u(1))*cos(u(2))')");
22     externalApp.eval("set_param('fsmkM','SimulationCommand','start')");
23
24     //Perform the simulation
25     do{
26       //Step the model
27       externalApp.eval("set_param('fsmkM','SimulationCommand','continue')");
28       do{
29         externalApp.eval("s=get_param('fsmkM','SimulationStatus')");
30         status=externalApp.getString("s");
31       }while (!status.equals("paused"));
32
33       //Get variables
34       value=externalApp.getDouble("y");
35       time=externalApp.getDouble("t");
36
37       System.out.println("time:"+time+" value:"+value);
38     } while (time<10);
39
40     //Stop the Simulink simulation
41     externalApp.eval("set_param('fsmkM','SimulationCommand','stop')");
42
43     //Finish the connection
44     externalApp.disconnect();
45   }
46 }
```

Listing 3.13: Computing a Function Using a Simulink model.

3.3.3 General process to simulate Simulink models from Java

The process described opens the way to simulate any Simulink model from a Java program. The process required to create interactive simulations using Simulink is now summarized in the following actions:

- Modify the original Simulink model to indicate that the simulation ends at the time `inf`.
- Modify the original Simulink model, adding blocks to read and write variables from the MATLAB workspace.
- Modify the original Simulink model, adding blocks to obtain the simulation time and also to pause the model by calling the function `set_param` to set the parameter `SimulationCommand` to the string `'pause'`.
- In the Java program, and after the MATLAB connection is started, the modified model has to be opened with the `open_system` function. Then, the variables required by the Simulink model have to be initiated. After that, the model is started setting the parameter `SimulationCommand` of the model to the string `'start'`.
- The simulation is performed, by calling three actions: advancing one integration step of the model, checking that the model has finished the integration process, and recovering the values of the variables than are written by the model to the MATLAB workspace.
- After performing the simulation of the Simulink model, stop the model by setting the parameter `SimulationCommand` to the string `'stop'`.
- Finally, close the MATLAB connection.

3.3.4 Specific modifications for integrator blocks

In principle, the previous process can be used to simulate any Simulink model from a Java program. This is especially true for static systems like the model **fsmk**. However, the simulation of dynamic systems can be different. In this kind of models, apart from a static description, differential or difference equations may concur, producing time-varying systems. Thus, contrary to static systems, the behaviour of dynamic systems

depend on the initial conditions. For this reason, in the bouncing ball example, if two identical balls are dropped from different heights (initial conditions), they will stop bouncing at different moments.

Hence, an interactive simulation of a dynamic system should be prepared to accept that users can change the initial conditions at any moment. Thus, for example, an interactive simulation of a bouncing ball could be paused by the user, who moves the ball to a different height to restart again the simulation from there.

The description of time-varying systems in Simulink can be done in many ways, but the most common method to formulate dynamic systems in Simulink is to add *Integrator* blocks to the model (see Figure 3.7).

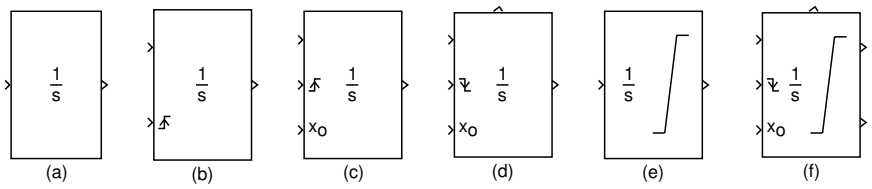


Figure 3.7: Various Integrator blocks. The appearance of each Integrator depends on the configuration defined in the dialog box.

An integrator block outputs the integral of its input at the current time step. Equation (3.1) represents the output of the block y as a function of its input u and an initial condition y_0, where y and u are vector functions of the current simulation time t. Take into account that the Integrator block outputs the initial condition at the beginning of the simulation and also when the Integrator is reset.

$$y(t) = \int_{t_0}^{t} u(t)dt + y_0 \tag{3.1}$$

In order to implement a correct manipulation of the Integrator block from the interaction point of view, it will be discussed now how to control from Java both the moment when an integrator is reset and the value of the initial condition.

Consider the model (named `integrator`) shown in Fig.3.8, which uses the simplest case of the Integrator block. Since the Integrator block is integrating a constant equal to 1, the output after the integration is a straight line with slope equal to 1. This line is shown in Figure 3.9.

To control the `integrator` model, it needs to be modified to obtain a new model (named `integratorM`) like Figure 3.10 shows. Now, the integrator accepts an external

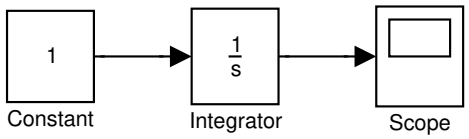


Figure 3.8: A model with an integrator.

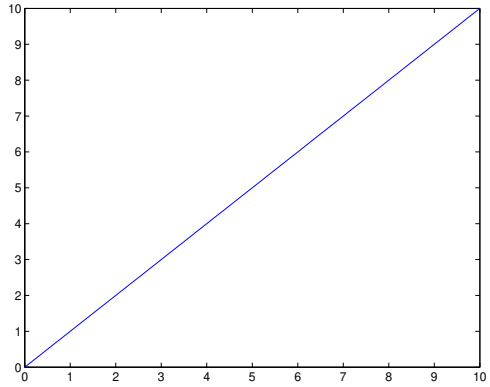


Figure 3.9: A plot of the model output.

reset and external initial condition. The two new inputs of the block integrator are connected to two MATLAB Fcn blocks to manipulate from the MATLAB workspace the reset moment and the initial condition. The two MATLAB Fcn are used to read the variables in the same way as the block named `FromWS` was used in Figure 3.6 to read the value of the variable `f`. The two variables read by the blocks MATLAB Fcn are `rst` and `ic`, which control the reset moment and the initial condition respectively. The output of the Integrator is sent to the MATLAB workspace (as variable `y`) using a block `To Workspace` similar to the block `toWS` used in the model of Figure 3.6a. Additionally, a sub model similar to the `stepCtrl` shown in Figure 3.6b is required to get the simulation time and to pause the Simulink model after each integration step.

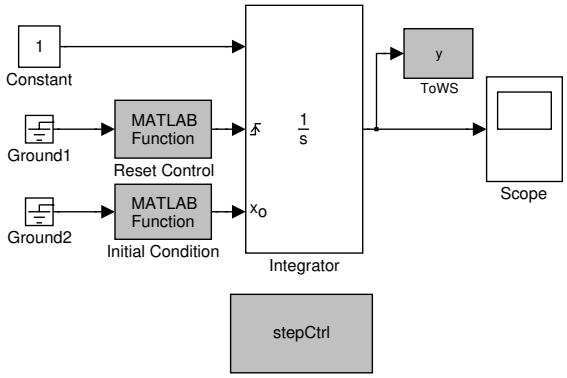


Figure 3.10: A modified model of the integrator.

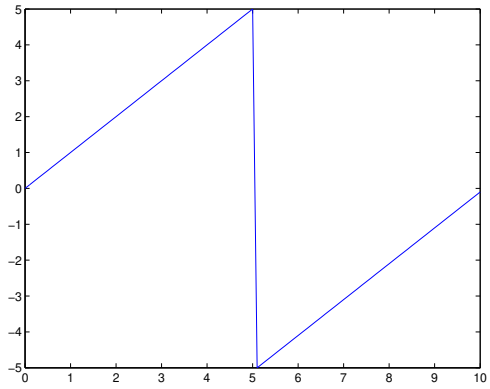


Figure 3.11: The output of the modified model.

After the modifications are done, the model `integratorM` can be used directly from a Java application to reset the integrator. Listing 3.14 shows the code of an application that resets the integrator at time=5. Note that the integrator is reset only once, because the reset is only triggered when a rising change in the variable `rst` (e.g., from -1.0 to

1.0) is detected. Fig.3.11 shows a plot of the output of the integrator. Observe that, at the beginning of the simulation, the initial condition (`ic`) was 0, and when the reset is triggered, the initial condition is set to -5 and therefore the state restarts from -5.0.

```
1  ...
2  //Prepare the simulation
3  externalApp.setValue("rst",-1.0);
4  externalApp.setValue("ic",0.0);
5  externalApp.eval("set_param('integratorM','SimulationCommand','start')");
6  //Perform the simulation
7  do{
8    //Step the model
9    externalApp.eval("set_param('integratorM','SimulationCommand','continue')");
10   do{
11     externalApp.eval("s=get_param('integratorM','SimulationStatus')");
12     status=externalApp.getString("s");
13   }while (!status.equals("paused"));
14   //Get Integrator's output and simulation time
15   output=externalApp.getDouble("y");
16   time=externalApp.getDouble("t");
17   //reset at time=5
18   if (time>=5){
19     externalApp.setValue("rst",1.0);
20     externalApp.setValue("ic",-5.0);
21   }
22   System.out.println("time:"+time+" output:"+output);
23 }while (time<10);
24 ...
```

Listing 3.14: Resetting an Integrator block from Java.

The simple case of the integrator described previously is however not common. Most of the dynamic systems in Simulink present more complex configurations, which means that sometimes the model itself uses the external reset and external initial condition to model events in the system. To correctly treat the events of the system with the events triggered by Java, the scheme of the Fig.3.12 has to be used.

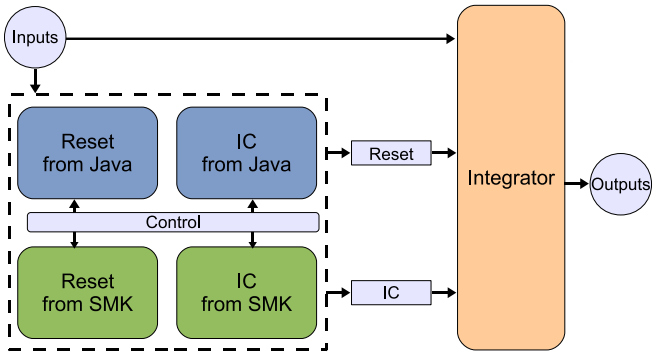


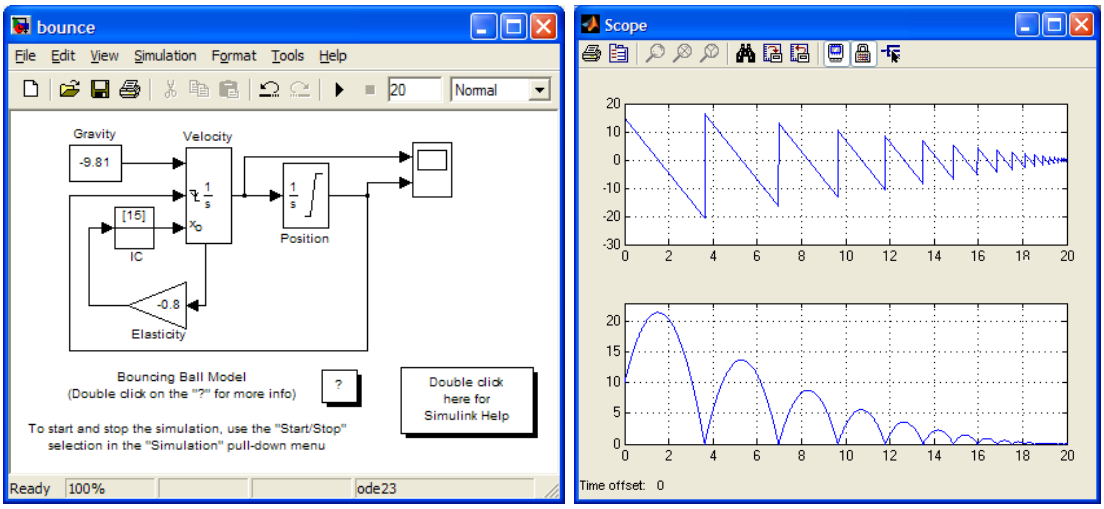
Figure 3.12: An scheme to treat Java and Simulink events in integrators.

The scheme requires the substitution of an integrator (with any configuration) by a sub model composed by an integrator with external reset(`Reset`) and external initial condition (`IC`). Both `Reset` and `IC` are computed according to the state of the signals obtained from Simulink and Java. The Simulink signals reflect the situation of the

original integrator in the Simulink model. The Java signals allow manipulating from Java to reset the integrator when, for instance, the user interacts with the Java simulation. Thus, both types of reset have been taken into account in the simulation. In the case that both types of reset are triggered at the same time, the Java reset should have priority over the Simulink reset in order to provide a good interaction experience to the end user. Obviously, the configuration of the original integrator (i.e. the limitation and the state port) has to be preserved in the new integrator.

3.3.5 An example of a dynamic system

An example of a non elastic bouncing ball will be used to show how the previous ideas are implemented. A rubber ball is thrown into the air with a velocity of 15 meters per second from a height of 10 meters. The position and velocity of the ball are shown in a Scope. This system uses a reset-integrator to change the direction of the ball as it comes into contact with the ground. Figure 3.13 shows the Simulink model of the bouncing ball system and the plots generated when the simulation is run.



a) b)

Figure 3.13: The simulation of a bouncing ball. a) Simulink model, b) Plots of the velocity and vertical position of the ball.

The model is quite simple. There are two continuous states, the velocity and the position. These states are available as outputs of the two integrators. The position is obtained by integrating the velocity, and the latter is obtained by integrating the gravity given by a Constant block. The integrator for the velocity also has another two inputs, that are used to reset the velocity to an initial state given by the third input. This allows

to model the bouncing of the ball when it reaches the ground (i.e., the position is zero). Since the ball is not elastic, the velocity of the ball is reset to a new velocity, which is computed multiplying the velocity *just before* the bouncing by an elastic constant equal to -0.8. The negative sign simply changes the vertical direction of the velocity. Note that the value of the velocity previous to the bouncing is taken from the state port (the second output) of the `Velocity` block in order to avoid algebraic loops. The integrator for position has its output limited to a minimum value of zero. This ensures that the position of the ball will never be negative. The Initial Condition block named `IC` outputs 15 when the simulation is started. This is used as the initial condition of the integrator `Velocity`. The Initial Condition block does not have any effect once the simulation is started.

A modified version of the dynamic system

Figure 3.14 shows the modified model of the bouncing ball. Since there were two integrators, both have been modified following the scheme shown in Figure 3.12. The two new sub models that represent the original integrators `Position` and `Velocity` are now `PositionM` and `VelocityM`, respectively. The necessary `stepCtrl` submodel to get the simulation time and to pause the Simulink model after each integration step has also been added.

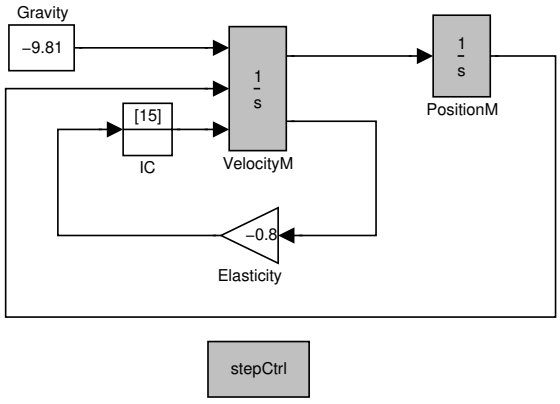


Figure 3.14: The modified model of the bouncing ball.

The submodel `PositionM` is shown in Figure 3.15. The submodel follows the scheme described in Figure 3.12. The new integrator has three inputs and one output. One input is used for the standard input of the signal to be integrated, and two inputs for supporting the external reset and external initial condition. The single output is used

for the standard output of the integrator, which gives the position of the ball. Note that the position is also written to the MATLAB workspace so that it can be read by the Java application. Observe also that the output limitation of the original integrator is replicated by the new integrator.

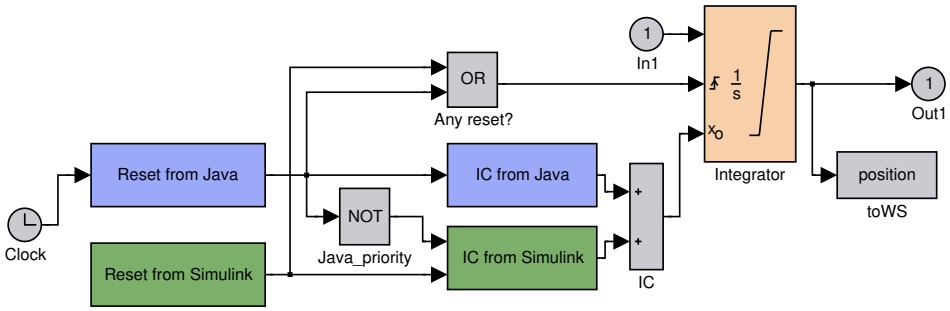


Figure 3.15: The `PositionM` submodel.

The standard input of the integrator is taken from the sole input, `In1`, of the submodel. This input is connected to the first output of the `velocityM` submodel, which is the velocity of the ball.

The external reset of the integrator is connected to the block named `Any reset?`, which implements an OR logic gate. This block sends out 1 (or the boolean true) if at least one of its two inputs are 1, otherwise the output is 0 (or the boolean false). The two inputs of the OR block are used to detect one of two possible resets. These inputs come from the submodels: `Reset from Java` and `Reset from Simulink`. If there is a reset, the initial condition is taken from either the submodel `IC from Java` or the submodel `IC from Simulink`. If both Simulink and Java reset are triggered at the same time, the block NOT, named `Java_priority`, is used to prioritize the Java reset.

See Appendix A for further details of the modifications of the other blocks of the Simulink model of the bouncing ball.

Using the modified dynamic system from Java

After the modifications are done, the modified bounce model, named `bounceM`, can be used directly from a Java application to reset the integrators. Listing 3.15 shows the code of an application that resets the integrators at time = 10. The reset implies setting the initial conditions to 10 for the position and 0 for the velocity.

```
1  ...
2  //Prepare the simulation
3  externalApp.setValue("rst",-1.0);
```

```
4   externalApp.setValue("ic_position",10.0);
5   externalApp.setValue("ic_velocity",15.0);
6   externalApp.eval("set_param('bounceM','SimulationCommand','start')");
7   //Perform the simulation
8   do{
9     //Step the model
10    externalApp.eval("set_param('bounceM','SimulationCommand','continue')");
11    do{
12      externalApp.eval("s=get_param('bounceM','SimulationStatus')");
13      status=externalApp.getString("s");
14    }while (!status.equals("paused"));
15    //Get Integrator's outputs and simulation time
16    position=externalApp.getDouble("position");
17    velocity=externalApp.getDouble("velocity");
18    time=externalApp.getDouble("t");
19    //reset at time=10
20    if (time>=10){
21      externalApp.setValue("rst",1.0);
22      externalApp.setValue("ic_position",10.0);
23      externalApp.setValue("ic_velocity",0);
24    }
25    System.out.println("time:"+time+" position:"+position+" velocity"+velocity);
26  }while (time<20);
27  ...
```

Listing 3.15: Simulating the modified bounce model with reset.

Figure 3.16 shows the plots of the position and velocity of the bouncing ball. The ball starts with a velocity of 15 meters per second and a height of 10 meters. A reset from Java is performed after 10 seconds. When this reset is triggered, the ball position and velocity are changed to 10 meters and 0 meters per second respectively.

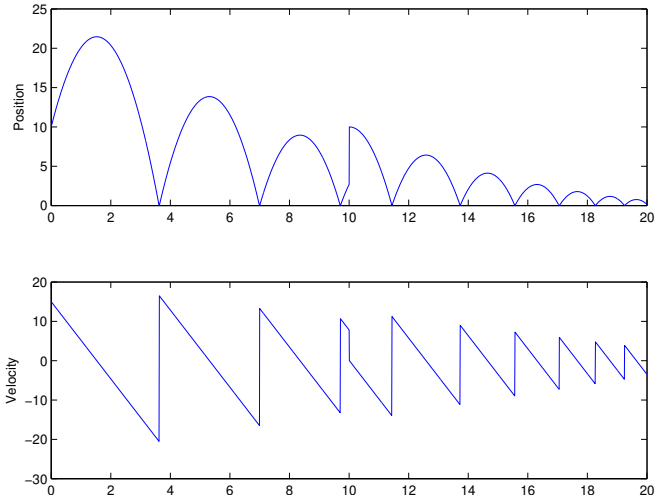


Figure 3.16: Plots of the position and velocity of the modified bounce model with reset at t=10.

Other situations

Using the described scheme to treat the Java and Simulink event in the integrators it should be enough to manipulate Simulink models with dynamic behaviour from Java. Fortunately there is a huge number of dynamic models that are described using integra-

tor blocks. However, there are also simulations or models described using other Simulink blocks such as `Transfer Function`, `State-Space` and `S-Function`. The first two types of blocks can always be replaced using integrators, since they describe linear models. However, the `S-Function` can describe much more complex models (e.g. non linear or hybrid models) which can not be formulated only using integrators. In that cases, users should try to modify these complex models to read and write in to MATLAB workspace the values to be controlled from the Java application.

3.3.6 Speeding up the simulations

Sometimes, the simulation of a Simulink model can be slow. This can be a consequence of diverse factors.

One of the most common situations is due to the many integration steps required by the simulation. This can produce a very slow simulation of the Java application because after every integration step, a pause is added by the submodel `stepCtrl` (see 3.6b) to update the connected variables, i.e., to read and write the variables from/to the MATLAB workspace.

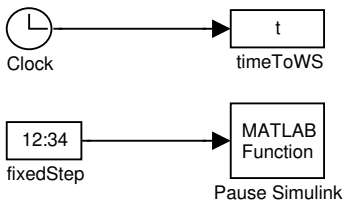


Figure 3.17: An alternative for the submodel `stepCtrl` to speed up the simulation.

Normally, the integration steps can be reduced by conveniently modifying the parameters of the solver of the Simulink model. For example, by selecting a Fixed-step solver with a bigger sample time. However, the many integration steps can also be due to the occurrence of many state or time events. This is specially true in the simulation of hybrid models. In this case, a good option is to update the variables not after two integration steps, but after a given time has elapsed. To do that, the submodel `stepCtrl` can be slightly modified by replacing the `Ground` block, named `Gr`, with the `Digital Clock`, named `fixedStep`, as Figure 3.17 shows. The `Digital Clock` block outputs the simulation time at the specified sampling interval to the `Pause Simulink` block. Thus, when the simulation time arrives to the block `Pause Simulink`, the simulation is paused. Incrementing the parameter `sample time` of the `Digital Clock` can then

speed up the simulation.

Other factors that normally affect the simulation speed are the animation blocks. This kind of blocks are useful when the simulation is run directly from Simulink, but not when it is going to be controlled from Java. These blocks are very time-consuming, which affects the performance of the simulation from Java. This is the case, for instance, with the `Scope` blocks of the models shown in Figure 3.13a) and Figure 3.5a). Probably the best option in this case is simply to delete them in the modified model, and to reconnect or eliminate the unlinked lines, to avoid warnings.

There can also be other factors that slow down the simulation of a Simulink model, and the modifications required probably depend on the particularities of each model. However, by implementing the recommendations above, the execution speed of the simulation can be considerably improved.

3.4 A direct implementation of `ExternalApp` for Simulink

Considering the way to control the Simulink model from Java, and the modifications required to simulate dynamic systems and to speed up the simulation, it is clear that the simulation can be performed by using only the Java class `MatlabExternalApp`. However, it would be preferable if these modifications required by the described process could be generated automatically depending only on the block's inputs and outputs to be manipulated, saving authors from doing all the necessary modifications to create the interactive simulation with Simulink models. This is the aim of the Java class `SimulinkExternalApp`.

3.4.1 The Java class `SimulinkExternalApp`

The `SimulinkExternalApp` Java class extends the `MatlabExternalApp` class. This implies that all methods of the `MatlabExternalApp` class are inherited by it. Thus, both the low and high level protocols are supported by this Java class `SimulinkExternalApp`. However, the methods implemented in the `MatlabExternalApp` class to support the high level protocol of the interoperate approach are overridden by the new class in order to provide a suitable simulation, manipulating correctly the linked variables, which are now connected with a block's input or output of the Simulink model. Listing 3.16 shows some of the methods overridden by the `SimulinkExternalApp` class.

```
1  ...
2  public class SimulinkExternalApp extends MatlabExternalApp{
3    ...
4    public SimulinkExternalApp (String mdlFile) {
5      ...
6    }
7    public boolean linkVariables(String cvar, String epath,
8                                 String etype, String eport){
9      ...
10   }
11   public String connect (){
12     ...
13   }
14   public void setValues(){
15     ...
16   }
17   public void getValues(){
18     ...
19   }
20   public void step (double dt) {
21     ...
22   }
23   public void synchronize(){
24     ...
25   }
26   static private final String conversionCommand = "...";
27   ...
28 }
```

Listing 3.16: `SimulinkExternalApp` class: Some of the methods implemented.

The constructor of the class `SimulinkExternalApp` requires a String input parameter, which provides the name of the Simulink model to be controlled from Java. There is a second constructor to support a fixed-time updated implementation (as described previously) to speed up the simulation.

Linking the variables

The implementation of the link between client and external variables is different from how the class `MatlabExternalApp` does it. Now, the connection is between a Java variable and an input, output, or a parameter of a Simulink block. This implementation requires a slight modification of the method `linkVariables` to support this Java variable-block's signal connection.

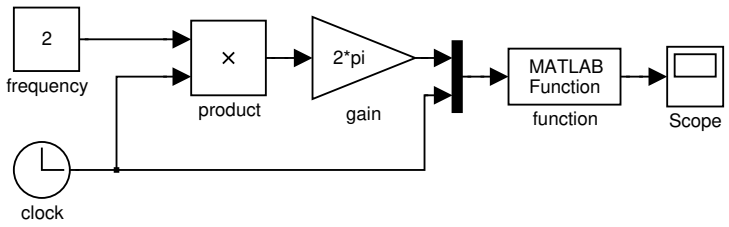


Figure 3.18: The model fsmk. The Simulink version of the evaluating function.

The method `linkVariables` has now four input parameters, the first one represents

the name of the client variable. The next three inputs represent the external connection. The input `epath` is the path of the route of a block inside the model. The parameter `etype` indicates the type of connection. The last parameter, `eport`, normally represents the port to be connected. For instance, the following instruction links the client variable `frequency` with the first input of the block `product` of the model `fsmk` (see Figure 3.18):

```
linkVariables("frequency","fsmk/product","in","1");
```

Note that the parameters `"in"` and `"1"` indicate the first input of the block `product`. The next instruction links the client variable `value` with the output of the block `function` of the model `fsmk`:

```
linkVariables("value","fsmk/function","out","1");
```

The method `linkVariables` can also be used to link a client variable with a block's parameter as the following instruction shows:

```
linkVariables("fsinus","fsmk/function","param","MATLABfcn");
```

The third input of `linkVariables("param")` indicates that the connection of the variable `fsinus` will be with a parameter (in this case `MATLABfcn`) of the block `function`.

Listing 3.17 shows the code to implement the `linkVariables` method. It is similar to the version of the class `MatlabExternalApp`. First, the code checks if the client variable `cvar` exists. If the variable exists, then the value of the variable is obtained depending on its type. The variable and its value are used to generate the string `evarInitValue`. This string creates and initializes the external variable. Note that the external variable is formed as the concatenation of the client variable and the `PREFIX` string. The value of this PREFIX is defined as the string `"Ejs_"`. Thus, the external variable resulting of the connection in `linkVariables("value","function","out","1")` is `Ejs_value`.

The `evarInitValue` string is concatenated to `initCommand`, which accumulates all the external variable initializations. Furthermore, the variable `initCommand` adds a register to the MATLAB structure `vars` to each external variable. The structure has four fields for each connection. The fields `vars.path`, `vars.fromto`, and `vars.port` have the same meaning as `epath`, `etype` and `eport` of `linkVariables`. The name of the external variable (product of the concatenation of the PREFIX and the client variable) is set in the `vars.name` field. The `vars` structure will be used later to modify automatically the

Simulink model according to the connection defined by `linkVariables`.

```
1  public boolean linkVariables(String cvar, String epath,
2                               String etype, String eport){
3    if (varContextObject==null) return (false);
4    int type;
5    //Search if the cvar exists
6    for (int i=0; i < varContextFields.length; i++) {
7      if (cvar.equals((varContextFields[i]).getName())) {
8        //Detect type
9        ...
10       //Get value of the client variable
11       String evarInitValue=PREFIX+cvar+"=";
12       try {
13         switch (type){
14           case DOUBLE:
15             evarInitValue=evarInitValue
16             +varContextFields[i].getDouble(varContextObject);break;
17           case ARRAYDOUBLE:
18             ...
19           case ARRAYDOUBLE2D:
20             ...
21           case STRING:
22             ...
23         }
24         //Add to string initCommand
25         initCommand=initCommand+evarInitValue+";"
26         +"vars.path{end+1,1}='"+epath+" ';"
27         +"vars.name{end+1,1}='"+PREFIX+cvar+" ';"
28         +"vars.fromto{end+1,1}='"+etype.toLowerCase()+" ';"
29         +"vars.port{end+1,1}='"+eport+" ';";
30         String[] _element={cvar,epath,etype,eport};
31         linkVector.addElement(_element);
32         return (true);
33       }catch (java.lang.IllegalAccessException e) {
34         ...
35         return (false);
36       }
37     }
38   }
39   return(false);
40 }
```

Listing 3.17: `SimulinkExternalApp` class: method linkVariables.

Modifying automatically the Simulink model

The modification of the model is done when the method `connect()` is called. This methods first opens the Simulink model and then executes a MATLAB code to transform the original Simulink model. This code is defined in the variable `conversionCommand`. Listing 3.18 shows part of the code used for the model transformation. Here, the `vars` structure previously defined is used to modify suitably each block. The modification depends on the `fromto` field, which can be: `in`, `out`, `param`, or `delete`. The first three options treat the type of connection defined by the `etype` input of the `linkVariables` method. The `delete` option is defined by the `deleteBlock` method, which allows the elimination of a specific block in the modified model.

If the connection is of `in` type then a `MATLAB Fcn` block, named `FromWS`, is added

to read the value of the variable defined in `vars.namei`. Note that the block `FromWS` is added in a submodel named with the value of `sub_in`. The first output of this submodel is connected to the required input block using the instruction:

```
add_line(parent,[sub_in,'/1'],[vars.path{i},'/',vars.port{i}]);
```

When the connection is of `out` type, it is necessary to check if the block is either a common block or an integrator. In the case of an integrator, the original integrator is replaced by a submodel as described before. The main actions here are to:

- Get the configuration of the original integrator. This determines the configuration of the new integrator and also the composition of the submodel `Reset from Simulink` and `IC from Simulink`.
- Create the submodel integrator. This puts all the blocks required inside the submodel that replaces the original integrator.
- Set the new integrator with the original configuration. Original values of some parameters, such as upper limit and lower limit, are set in the new integrator.
- All functional blocks, such as those required to implement `Reset from Simulink`, `IC from Simulink`, `Reset from Java`, `IC from Java`, are added. Note also that a `To Workspace` block is added to send to the MATLAB workspace the output of the new integrator, which is written in the variable given by `vars.namei`.

The case of a common block is much easier to deal with, since this case is similar to the `in` option. Here a block `To Workspace` is added to send to the MATLAB workspace the output of connected block. The output obtained is written in the variable given by `vars.namei`. Note also that the block `ToWS` is added in a submodel named by the value of `sub_out`.

The option `delete` simply deletes the block defined by `vars.pathi` using the MATLAB function `delete_block`.

If the connection is of type `param`, a string `cmd` is created. This string forms the MATLAB function `set_param`, which will be executed after each integration step to update the value of the required block parameter.

After all the blocks are processed, the submodel `stepCtrl` is added to the modified model. Here the commands to pause the Simulink model after each integration step

and to update the defined block parameters are incorporated in the `MATLAB Fcn` block, named `Pause Simulink`.

```
...
%Process the structure vars
for i=1:length(vars.path)
  switch vars.fromto{i}
    case 'in'
     ...
     %Create submodel in
     add_block('built-in/subsystem',[parent,'/',sub_in]);
     add_block('built-in/matlabfcn',[parent,'/',sub_in,'/FromWS']);
     set_param([parent,'/',sub_in,'/FromWS'],'MATLABFcn',vars.name{i});
     add_block('built-in/outport',[parent,'/',sub_in,'/OUT']);
     add_block('built-in/ground',[parent,'/',sub_in,'/G']);
     ...
     add_line(parent,[sub_in,'/1'],[vars.path{i},'/',vars.port{i}]);
    case 'out'
     if isIntegrator(vars.path{i})
      %Get configuration of the original integrator
      flag_int_er=get_param(vars.path{i},'ExternalReset');
      flag_int_lu=get_param(vars.path{i},'UpperSaturationLimit');
      ...
      %Create submodel integrator
      add_block('built-in/subsystem',sub_integrator);
      add_block(['built-in/',int_type],[sub_integrator,'/I']);
      %Set the new integrator with original configuration
      set_param([sub_integrator,'/I'],'UpperSaturationLimit',flag_int_lu);
      ...
      %Add functional blocks to submodel integrator
      add_block('built-in/toworkspace',[sub_integrator,'/toWS']
      set_param([sub_integrator,'/toWS','VariableName',vars.name{i});
      if not(strcmp(flag_int_er,'none'))
       add_block('built-in/subsystem',[sub_integrator,'/reset_smk']);
      ...
     else
      %Create submodel out
      add_block('built-in/subsystem',[parent,'/',sub_out]);
      add_block('built-in/inport',[parent,'/',sub_out,'/IN']);
      add_block('built-in/toworkspace',[parent,'/',sub_out,'/ToWS'])
      set_param([parent,'/',sub_out,'/ToWS'],'VariableName',vars.name{i});
      ...
     end;
    case 'delete'
      delete_block(vars.path{i});
      ...
    case 'param'
     cmd=[cmd,'set_param(''',vars.path{i},''',''',
          vars.port{i},''',[''[''',num2str(',vars.name{i},')','']''',]);'];
     ...
  end;
end;
%Create submodel stepCtrl
...
cmd=[cmd,'set_param(''',model,''',',''',
     'SimulationCommand',''',',',''',Pause',''',');'];
cmd=['eval(''',strrep(cmd,'''',''''''),''');'];
%Add pause command
set_param([name_stepCtrl,'/Pause Simulink'],'MATLABFcn',cmd);
...
```

Listing 3.18: MATLAB code to modify the original model.

Controlling the simulation of the Simulink model

After this automatic modification of the model, the simulation is ready to be executed by calling the method `step`. A part of the implementation of this method is shown

in Listing 3.19. Here the methods `setValues` and `getValues` are called similarly to the implementation of `step` in the class `MatlabExternalApp`. The difference in this implementation comes from the simulation of the Simulink model. First the method `step` checks the variable `resetIC` in order to execute a reset of the integrators. The value of `resetIC` is set to true calling the method `synchronize()`. As described before, the reset is performed by modifying from zero (or positive) to a negative value the MATLAB variable `rst`.

```
1  ...
2  public void step (double dt) {
3    //Set all external variables
4    setValues();
5
6    //Step the model
7    ...
8    if (resetIC) {
9      eval("rst = 1 - rst");
10     startRequired = true;
11     resetIC = false;
12   }
13   ...
14   if (startRequired) {
15     startRequired = false;
16     eval("set_param ('"+theModel+"', 'SimulationCommand','Start')");
17     ...
18   }
19   for (int i=0,times=(int) dt; i<times; i++) {
20     eval("set_param ('"+theModel+"', 'SimulationCommand','Continue')");
21     do{
22       eval("s=get_param('bounceM','SimulationStatus')");
23       status=getString("s");
24     }while (!status.equals("paused"));
25   }
26
27   //Set all client variables
28   getValues();
29 }
30 ...
```

Listing 3.19: `MatlabExternalApp` class: Implementation of method step.

The `step` method also checks if it is the first time that the model is stepped. In this case, the model has first to be started using the parameter `SimulationCommand`.

Once the model has been started, it is stepped by modifying properly the parameter `SimulationCommand` to `Continue`. This advances the model one integration step. Note that, after this, it is necessary to check if the Simulink model is paused to perform again an integration step. The number of steps performed is given by the input parameter `dt`.

After the description of the main methods of the Java class `SimulinkExternalApp`, the next subsection presents a simple example of the use of this class.

3.4.2 Using the class SimulinkExternalApp from a Java program

In this subsection, the Java class `SimulinkExternalApp` will be used to create simple interactive Java simulations with Simulink models.

A first example

The first example is described in Listing 3.20. Here the model `fsmk`, shown in Figure 3.18, to evaluate a function using Simulink is again considered, but now using the Java class `SimulinkExternalApp`. First, an instance of the `SimulinkExternalApp` object is created using the constructor. Note that in the constructor the input parameter defines the Simulink model (in this case `fsmk.mdl`) to be used. After that, the client object is set in order to connect the client and external variables. Then, the `linkVariables` method connects the Java `frequency` variable with the first input of the `product` block. The `linkVariables` method also connects the Java `value` variable with the first output of the `function` block. The `linkVariables` method connects the Java variable `time` with the parameter `time` of the Simulink model `fsmk`.

After linking the variables, the connection with the external application is started. The simulation is performed by executing continuously the `step` method. When the variable time is equal to 10, the simulation ends and the connection with the application is finished. Although, in this simple example, the performance of the simulation is not a problem, note that the block `Scope` can be eliminated using `deleteBlock("fsmk/Scope")` to speed up the simulation.

```
1  ...
2    public evaluatingFunctionSimulink(){
3      //Create a Simulink connection
4      SimulinkExternalApp externalApp = new SimulinkExternalApp("fsmk.mdl");
5
6      //Set location of the Java variables
7      externalApp.setClient(this);
8
9      //Link Java and Simulink variables
10     externalApp.linkVariables("frequency","fsmk/product","in","1");
11     externalApp.linkVariables("value","fsmk/function","out","1");
12     externalApp.linkVariables("time","fsmk","param","time");
13
14     //Start the connection
15     externalApp.connect();
16
17     //Perform the simulation
18     do{
19       //Step the model
20       externalApp.step(1);
21
22       System.out.println("time:"+time+" value:"+value);
23     } while (time<10);
24
```

```
25        //Finish the connection
26        externalApp.disconnect();
27      }
28    }
```

Listing 3.20: Second version *Computing a Function Using a Simulink model.* of Listing 3.13 using the `SimulinkExternalApp` Java class.

A second example

The second example is described in Listing 3.21. Here, the model of the bouncing ball, shown in Figure 3.13a, is used again but now with the `SimulinkExternalApp` Java class. The application starts creating an instance of the `SimulinkExternalApp` class. Then, the client object is set with the `setClient` method. This informs `SimulinkExternalApp` where the Java variables are located. After that, the process to link variables starts. The Java variable `position` is linked to the first output of the Integrator block `Position`. The Java variable `velocity` is connected to the first output of the Integrator block `Velocity`. Finally, the Java variable `time` is linked to the parameter `time` of the model `bounce`. Observe that the parameter time is always available for any Simulink model using the Java class `SimulinkExternalApp`. After the connection between client and external application is established, the simulation is executed by calling the `step` method. Note that an integrator reset is executed at time = 10. At this time, the position and the velocity of the ball are set to the values 10 and 0 respectively. After 20 seconds of simulation, the execution ends and the connection with the external application finishes.

```
1   ...
2   //Create a Simulink connection
3   SimulinkExternalApp externalApp = new SimulinkExternalApp("bounce.mdl");
4
5   //Set location of the Java variables
6   externalApp.setClient(this);
7
8   //Link Java and Simulink variables
9   externalApp.linkVariables("position","bounce/Position","out","1");
10  externalApp.linkVariables("velocity","bounce/Velocity","out","1");
11  externalApp.linkVariables("time","bounce","param","time");
12
13  //Start the connection
14  externalApp.connect();
15
16  boolean firstReset=true;
17  //Perform the simulation
18  do{
19    //Step the model
20    externalApp.step(1);
21    //reset at time=10
22    if (time>=10 && firstReset){
23      position=10;velocity=0;
24      externalApp.synchronize();
25      firstReset=false;
26    }
27    System.out.println("time:"+time+" position:"+position+" velocity"+velocity);
```

```
28  }while (time<20);
29
30  //Finish the connection
31  externalApp.disconnect();
32  ...
```

Listing 3.21: Simulating the bounce model.

Obviously, more elaborate examples can be obtained without too much effort. For instance, similar functionality to the one described in the Listing 3.12 could be used to plot the position and velocity of the bouncing ball. In addition, some visual elements such as sliders and buttons could be considered to provide a natural way to perform a reset of the ball.

3.5 Remote interfacing MATLAB and Simulink with Java

To extend the capabilities of the local link between Java and MATLAB/Simulink, a remote link has been also implemented. This remote link was developed following the design proposed by the communication protocol of the interoperate approach.

The design of the remote link implies the implementation of two components of software: the client and server side implementation of the communication protocol. The client side provides all the functionality required by the communication protocol, but instead of controlling directly MATLAB/Simulink, the client implementation executes remote calls to the server implementation. This server side accepts and replies to client calls by using all the features developed in the local connection with MATLAB/Simulink.

The server side implementation is supported by a Java software tool named Java Internet MATLAB (JIM for short). Similarly to the local link, the library JMatLink is used to control MATLAB. The client side is coded as an extension, with new functionalities, of the local Java package developed in the local link among MATLAB and Java. This new and wider Java package is called **JIMC** (Department of Computer Science and Automatic Control, UNED 2010*b*).

3.5.1 The software JIM server

In order to support the remote calls from JIMC, the JIM server uses the Java class `MatlabExternalApp` to control MATLAB. The normal state of the server is to wait for a remote connection from the client application. When a remote call arrives through the network, the server executes the corresponding method of the `MatlabExternalApp`

class. Listing 3.22 shows a piece of code of the JIM server that supports some remote calls such as `eval`, `setDouble`, and `getDouble`.

```
1   ...
2   // Evaluate a command
3   if (RemoteFunction.equalsIgnoreCase("eval")) {
4     String command = (String) bufferInput.readUTF();
5     matlab.engEvalString(id, command);
6   }
7
8   // Set a double
9   if (RemoteFunction.equalsIgnoreCase("setValueDouble")) {
10    String variable = (String) bufferInput.readUTF();
11    double valueDouble = bufferInput.readDouble();
12    matlab.engPutArray(id, variable, valueDouble);
13  }
14
15  // Get a double
16  if (RemoteFunction.equalsIgnoreCase("getDouble")) {
17    String variable = (String) bufferInput.readUTF();
18    double valueDouble = matlab.engGetScalar(id, variable);
19    bufferOutput.writeDouble(valueDouble);
20    bufferOutput.flush();
21  }
22  ...
```

Listing 3.22: Part of the code that implements JIM server.

Since the main idea is to manipulate simulations over networks, the remote link is based on a client/server implementation of a TCP connection. This network protocol stands for Transmission Control Protocol, and is widely used for well-known Internet applications such as the Web, FTP, e-mail, Telnet, etc. Unlike UDP (User Datagram Protocol), TCP offers a reliable transmission, flow control, and congestion control.

On the Internet, information is transmitted in small pieces of data called packets. These packets can suffer errors during the transmission or even arrive to the final destination in disorder. To avoid these problems, TCP uses a mechanism of acknowledgements and retransmissions to make sure that the end-to-end communication is free of errors and the packets are received in the original order.

TCP also uses an end-to-end flow control protocol to avoid having the sender send data too fast for the TCP receiver to reliably receive and process it. Having a mechanism for flow control is essential in an environment where machines of various network speeds communicate (Kurose & Ross 2009). For example, if the server sends data much faster than a slow client can process, the client must regulate data flow so as not to be overwhelmed. This way, the flow control protocol matches the rates at which the client consumes and the server produces data.

The final main aspect of TCP is congestion control. TCP uses a number of algorithms to achieve high performance and avoid *congestion collapse*, where network performance can fall by several orders of magnitude. These mechanisms control the rate of data entering the network, keeping the data flow below a rate that would trigger collapse.

Despite the fact that congestion control can occasionally reduce the transmission rate, the selection of TCP to support the remote manipulation of simulations is preferred especially because it provides reliable transmission and flow control, which must be implemented in the case of UDP.

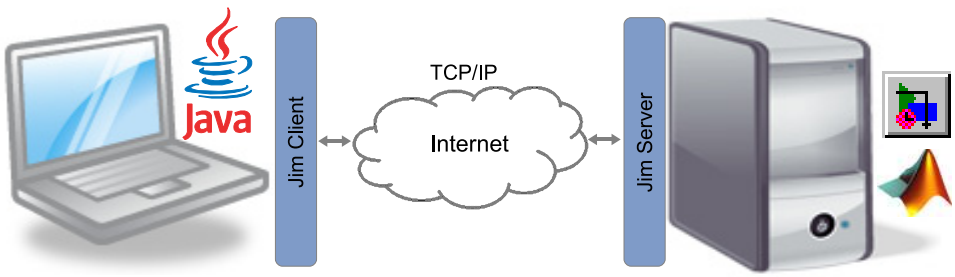


Figure 3.19: The interoperate approach for a remote link.

The scheme of the remote link is presented in Figure 3.19. Note that the communication protocol is implemented in the client and server sides. The approach encapsulates the network in such a way that only network delays could be appreciated by end users as differences from a simulation using a local link. In addition, from the design point of view, the creation of the interactive simulations for both local and remote links is similar. Hence, if network delays are negligible, then authors can use the interoperate approach independently of the type of the link selected to deploy the interactive simulation. However, if network delays are high, then the simulation could perform poorly. For this reason, to avoid this undesired effects, there are two versions of the remote link implemented: **synchronous** and **asynchronous**.

Graphical User Interface of JIM

The graphical user interface of JIM is depicted in Figure 3.20. In the user interface, it is possible to use the following options to set a TCP/IP link between JIM and the Java program and MATLAB/Simulink (default values are in bracket):

- **Service Port**: This option sets the socket port number of the TCP/IP link at the server side (2005).

- **Buffer In Size**: Maximum size of the input buffer for both server and client applications (1024).
- **Buffer Out Size**: Maximum size of the output buffer for server and client applications (1024).
- **Work Directory**: Directory where the user's MATLAB/Simulink files are downloaded (\JimWD). The directory is relative to the path of the MATLAB installation.
- **Max MATLAB Sessions**: Maximum number of MATLAB sessions allowed (5).
- **Allow OS Functions**: If this option is checked, then a remote user can use MATLAB functions (e.g. DOS or Bang Operator "!"). These functions call upon the shell to execute the given command for Windows systems (not checked).
- **Allow External Files**: If this option is checked, then a remote user can use his/her own MATLAB/Simulink files (checked).
- **Log File**: If this option is checked, then the server saves a log file with the remote user activities (checked).
- **Authenticate**: If this option is checked, then the remote user must authenticate to connect the remote MATLAB (not checked). The database with the information about authorized users must be indicated if the authentication is required.
- **User**: The username of the administrator of the database(*blank*).
- **Pwd**: The password of the administrator of the database(*blank*).

The service port number indicates the socket port where the JIM server provides the TCP connection. This number has to be between 0 and 65535, and it should be chosen carefully to avoid any port conflict with another service. A list of well-known service port numbers can be found in (Kurose & Ross 2009). Normally, a number above 1024 should not cause any conflict.

The buffer sizes affect indirectly the flow and congestion control of the TCP protocol. Small buffers can reduce drastically the transmission rate of the server, introducing delays artificially in the remote connection. Big buffers do not have, in principle, negative

effects on the performance of the network communication and are limited only by the size of the memory of the client and server computers. However, take into account that a big buffer can be only an artificial number since the congestion control can automatically reduce the transmission rate if any collapse on the network is detected.

Normally, to execute a remote simulation, some M or Simulink files on the server side are needed. These files must be allocated to the work directory defined in JIM's options. By default, the path to this directory is `\JimWD`, which is relative to the path of the MATLAB installation (e.g. `C:\MATLAB`). The files can be uploaded from the client to server over the TCP connection if the option *Allow External Files* is checked.

When a remote user connects to the server, an exclusive MATLAB session is opened. This gives an independent workspace to each user. However, too many MATLAB sessions can decrease the server performance drastically. Therefore, a maximum number of sessions is set by JIM. Obviously, this number depends on the capacity of the computer where the server is running.

MATLAB allows users to access directly the shell functions of the Operating System. In principle, this opens the applications widely, but can also be dangerous for the integrity of the server. For that reason, a custom filter has been included in the options of JIM. However, instructors must consider this filter a basic barrier, since MATLAB has no built-in functionality yet to avoid access the shell. To decrease this risk, instructors can also use the authentication feature of JIM.

If the authentication is on, remote users must send their credentials (i.e., username and password) to the server. JIM will check in a database if the received credentials are correct. When the authentication is successful, JIM also checks if the user has a reservation to use the server. The booking defines a slot of time in which the authenticated user can access the remote service. Both credentials and booking information have to be located in a MySQL database system. The database consists of at least one table named `accesslist` with the following fields: `username`, `password`, `starttime`, `endtime`, and `valid`. The third and fourth fields define the booking slot. All fields are of the String type.

Finally, JIM provides a message area where the main actions of the server are saved. This information can be highly detailed if the option `log` is selected as `full`.

The server can be stopped at any time by the server administrator to change any of the options. Obviously, if a remote user is using JIM, then the link will be broken and the user must connect the client application again.

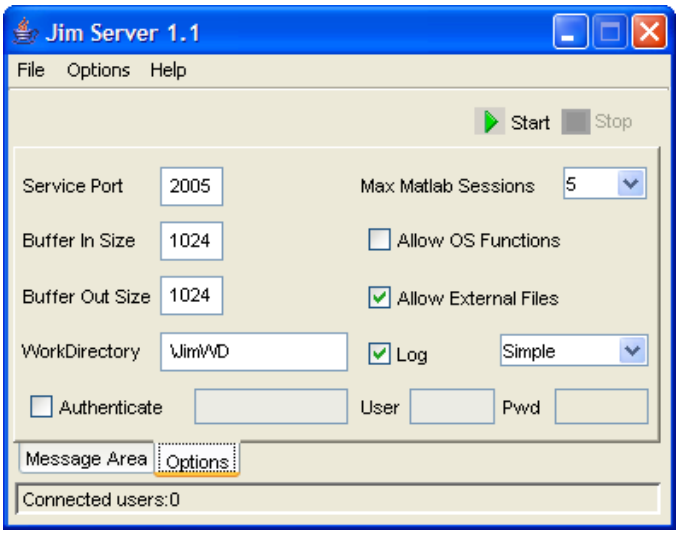


Figure 3.20: Graphical user interface of JIM.

3.5.2 Implementation of the communication protocol for remote links

In order to implement the interoperate approach for remote experimentation, one Java class has been implemented for MATLAB and Simulink.

Following the requirements of the communication protocol, the remote simulations with MATLAB and Simulink accept two versions of the remote link: **synchronous** and **asynchronous**. Both modes of the remote link have been implemented to support the connection with MATLAB and Simulink.

Next blocks describe both MATLAB and Simulink implementations of the Java interface `ExternalApp` to support the communication protocol.

The Java class `RMatlabExternalApp`

Listing 3.23 shows the three methods required to initiate the remote simulation: The constructor and the two methods for starting the connection.

```
1  ...
2  public class RMatlabExternalApp implements ExternalApp{
3    ...
4    public RMatlabExternalApp(String config) {
5      //Process config
6      String[] params=config.split(":");
7      ...
8      if (params[0].equals("matlabas")) asynchronousMode=true;
9      SERVICE_IP = params[1];
10     SERVICE_PORT = Integer.parseInt(params[2]);
```

```
11        ...
12      }
13
14      public boolean connect(){
15        result=connect("","");
16        ...
17        return result;
18      }
19
20      public boolean connect (String user, String pwd){
21        ...
22        if (jimTCP!=null){
23          if (!jimTCP.isClosed()) return true;
24        }
25        jimTCP = new java.net.Socket(SERVICE_IP, SERVICE_PORT);
26        bufferInputTCP = new DataInputStream(
27         new BufferedInputStream(jimTCP.getInputStream()));
28        bufferOutputTCP = new DataOutputStream(
29         new BufferedOutputStream(jimTCP.getOutputStream()));
30        ...
31        //Check if authorization is required
32        Boolean authorization=bufferInputTCP.readBoolean();
33        //Authentication
34        int result=2;  //result of authorization
35        long rtime=-1; //time available
36        if (authorization){
37          bufferOutputTCP.writeUTF(user);
38          bufferOutputTCP.writeUTF(pwd);
39          bufferOutputTCP.flush();
40          result=bufferInputTCP.readInt();
41          rtime=bufferInputTCP.readLong();
42        }
43        ...
44        //User authorized
45        if (result==2) {
46          ...
47          return true;
48        }
49        //User not authorized
50        ...
51        disconnect();
52            return false;
53      }
54    ...
55    }
```

Listing 3.23: `RMatlabExternalApp` class: Some of the methods to initiate the remote operation.

The first pre-requisite in the remote connection is to define the IP (`SERVICE_IP`) and the Port number (`SERVICE_PORT`) of the JIM server. This is done by using the constructor of the `RMatlabExternalApp` class, as the following example shows:

```
RMatlabExternalApp("<matlab:62.204.192.27:2005>")
```

In this case the IP is 62.204.192.27 and the Port is 2005.

The constructor of `RMatlabExternalApp` is also used to set the mode of remote link required, i.e., synchronous or asynchronous. For instance, in the previous example, the link selected was synchronous, indicated by the `matlab` keyword. To select an asynchronous version, the keyword required is `matlabas`, as the following example shows:

```
RMatlabExternalApp("<matlabas:62.204.192.27:2005>")
```

Once the IP and Port numbers are given, the TCP connection with the JIM server can be executed by using the `connect` method. This method accepts a simple connection between the client and the external application. However, it is also possible to support an authorized connection by indicating the username (`user`) and password(`pwd`) as input parameter when the method `connect` is invoked.

The TCP connection creates a Socket with an output (`bufferOutputTCP`) and input (`bufferInputTCP`) buffers. Both buffers are used to communicate `RMatlabExternalApp` with the JIM server. The output buffer is used by the `RMatlabExternalApp` class to send the request message to the server, and the input buffer is used to receive the response from the server. For instance, the following instruction gets a Boolean from the input buffer, which indicates if the server needs an authorized connection or not:

```
Boolean authorization = bufferInputTCP.readInt()
```

If the server requires an authorized connection, then the `RMatlabExternalApp` class has to send the username and the password to the JIM server, using the output buffer as follows:

```
bufferOutputTCP.writeUTF(user)
bufferOutputTCP.writeUTF(pwd);
```

Note the use of `bufferOutputTCP.flush()` after sending the username and password to the server. This forces the data out onto the network. Otherwise, the Java class `RMatlabExternalApp` could wait forever for a response (i.e., whether or not the user is authorized) from the server.

Once the connection is started, the Java application can use the communication protocol to manipulate the remote MATLAB. Some implementations of the low-level protocol are shown in the Listing 3.24. Review Listing 3.22 to analyse how the remote calls of the methods `eval`, `setValue`, and `getDouble` are processed in the JIM server.

Observe that all methods implemented in the `RMatlabExternalApp` class use the input and output buffers to exchange information with the server. This process is always divided in the following steps:

- Send an identification number (i.e., the value `ID`).
- Send the name of the method.

- Send (if required) the input parameter of the method.

- Get (if required) the result of the remote execution of the method.

```
1  ...
2  public void eval(String command){
3    bufferOutputTCP.writeInt(ID);
4    bufferOutputTCP.writeUTF("eval");
5    bufferOutputTCP.writeUTF(command);
6    bufferOutputTCP.flush();
7    ...
8  }
9
10 public void setValue(String name, double value) {
11   bufferOutputTCP.writeInt(ID);
12   bufferOutputTCP.writeUTF("setValueDouble");
13   bufferOutputTCP.writeUTF(name);
14   bufferOutputTCP.writeDouble(value);
15   bufferOutputTCP.flush();
16   ...
17 }
18
19 public void setValue(String name, double value, boolean flushNow) {
20   bufferOutputTCP.writeInt(ID);
21   bufferOutputTCP.writeUTF("setValueDouble");
22   bufferOutputTCP.writeUTF(name);
23   bufferOutputTCP.writeDouble(value);
24   if (flushNow) bufferOutputTCP.flush();
25 }
26
27 public double getDouble(String name) {
28   if (!isSynchronized) return getDoubleAS();
29   if (asynchronousMode) haltUpdate(true);
30   ...
31   bufferOutputTCP.writeInt(ID);
32   bufferOutputTCP.writeUTF("getDouble");
33   bufferOutputTCP.writeUTF(name);
34   bufferOutputTCP.flush();
35   double valueDouble = bufferInputTCP.readDouble();
36   return (valueDouble);
37 }
38
39 protected double getDoubleAS() {
40   double valueDouble= bufferInputTCP.readDouble();
41   return(valueDouble);
42 }
43 ...
```

Listing 3.24: `RMatlabExternalApp` class: Some of the methods implemented of the low-level protocol.

The methods `eval` and all the methods `setValue` execute the steps of the process described before. The method to evaluate a command and all the methods to set values have two versions. One standard version (such as `setValue(String name, double value)`) and another version that controls the functionality provided by the `bufferOutputTCP.flush()` method. This latter version allows the accumulation of more than one remote calls in just one package, which decreases network delays when for instance various `setValue` methods are executed at the same time.

There are also two versions for all the methods to get data from the server (such as `getDouble`). One version to support the synchronous link (such as the `getDouble` itself) and another version to support the asynchronous link (such as the `getDoubleAS`). The first version implements the steps described before, but the second version only gets from the input buffer the results of the remote execution. All the second versions are only called internally by the corresponding first version when the variable `isSynchronized` is `true`. This variable is used in asynchronous mode to indicate when a synchronization between client and remote applications has been performed. Note that if the variable `isSynchronized` is `false`, but the mode is asynchronous, then the method `haltUpdate` is executed. This method clears the input buffer of old data sent by the server. Take into account that, on some occasions, the server produces data faster than the client can consume. The synchronization between client and server applications is executed by calling the method `synchronize`, which is shown in the Listing 3.25.

```
1  ...
2  public void synchronize(){
3    isSynchronized=true;
4  }
5
6  public  void step (double dt){
7    if (asynchronousMode)
8      stepAS(dt);
9    else
10     stepSYN(dt);
11 }
12
13 protected void stepAS (double dt){
14   if (isSynchronized){
15     setValues();
16     haltUpdate(true);
17     ...
18     bufferOutputTCP.writeInt (ID);
19     bufferOutputTCP.writeUTF ("stepMatlabAS");
20     bufferOutputTCP.writeUTF (externalVars);
21     bufferOutputTCP.writeUTF (command);
22     bufferOutputTCP.writeInt ((int)dt);
23     bufferOutputTCP.writeInt(packageSize);
24     bufferOutputTCP.flush();
25     isSynchronized=false;
26   }
27   getValues();
28 }
29
30 protected void stepSYN (double dt) {
31   setValues();
32   int steps=(int)dt;
33   for (int i=0;i<steps;i++)
34     eval(command,false);
35   getValues();
36 }
37 ...
```

Listing 3.25: `RMatlabExternalApp` class: Some of the methods implemented of the high-level protocol.

Listing 3.25 also shows the `step` method. Note that two types of steps can be executed depending on the mode of the remote link selected. If the mode is asynchronous, then the method `stepAS` is called, otherwise the method `stepSYN` is executed. Note that the latter method is almost the same as in the implementation of the `step` method of the local connection with MATLAB (see Listing 3.10). The only difference is the control over the `bufferOutputTCP.flush()` method to avoid network delays when `dt` is greater than 1.

The `stepAS` method is radically different. Here, the external variables are set (by using `setValues()`) only the first time. Besides, this first time is used to communicate to the server the command to be executed by MATLAB. Note also that a string named `externalVars` is sent to the server. This string has all the names of the external variables to be collected. A value of the `externalVars` could be:`"time,value"`. This string is created each time that the `linkVariables` method is called.

Using the information provided by the `stepAS` method, the server only takes care of executing the value of the string `command` and returning the variables declared in the string `externalVars`.

After the main methods of the Java class `RMatlabExternalApp` have been described, the following sections show how to use it.

Using the class `RMatlabExternalApp`

As the first example for the Java class `MatlabExternalApp`, Listings 2.5 and 2.8 of Chapter 2 are selected to show how `RMatlabExternalApp` is used. Those listing can be easily transformed to be used with the class `RMatlabExternalApp` by just replacing the line:

```
MyExternalApp externalApp = new MyExternalApp();
```

With the following line:

```
ExternalApp externalApp =
  new RMatlabExternalApp("<matlab:ipserver:portserver>");
```

Where the strings `ipserver` and `portserver` are the IP address and the Port number of the JIM server. The execution of this new example produces exactly the same output for the evaluation of the function as before.

In order to use the asynchronous link, Listing 2.8 has to be modified by replacing the line:

```
MyExternalApp externalApp = new MyExternalApp();
```

with the following line:

```
ExternalApp externalApp =
  new RMatlabExternalApp("<matlabas:ipserver:portserver>");
```

However, in this asynchronous version it is also necessary to modify the command to be evaluated in the remote MATLAB. So the line:

```
externalApp.setCommand("y=sin(2*pi*f*t)*cos(t)");
```

is changed to the following line:

```
externalApp.setCommand("y=sin(2*pi*f*t)*cos(t),t=t+0.1");
```

Thus the variable `t` (the time) is increasing directly in the remote MATLAB. Obviously, to capture any change in the variables of the client application (e.g. the command), the synchronize method has to be executed.

The Java class `RSimulinkExternalApp`

Listing 3.23 shows some methods required to initiate the remote simulation: The constructor, and the methods for starting the connection.

```
1   ...
2   public class RSimulinkExternalApp extends RMatlabExternalApp{
3     ...
4     public RSimulinkExternalApp(String config) {
5       //Process config
6       String[] params=config.split(":");
7       ...
8       if (params[0].equals("matlabas")) asynchronousMode=true;
9       SERVICE_IP = params[1];
10      SERVICE_PORT = Integer.parseInt(params[2]);
11      model = params[3];
12      ...
13    }
14
15    public boolean connect (String user, String pwd){
16      ...
17      if (jimTCP!=null){
18        if (!jimTCP.isClosed()) return true;
19      }
20      jimTCP = new java.net.Socket(SERVICE_IP, SERVICE_PORT);
21      ...
22      //Check if authorization is required
23      ...
24      //User authorized
25      if (result==2){
26        ...
27        //Check the model file
```

```
28          File modelFile = new File(model);
29          if (!modelFile.exists()){
30           if (!remoteFileExist()) {
31             System.out.println("Error Model " + model +
32               " doesn't exist in Local or Remote Place");
33             disconnect();
34             return false;
35            }
36          }else{
37            createModel();
38          }
39          //Open the model
40          openModel();
41          return true;
42        }
43        //User not authorized
44        ...
45        disconnect();
46            return false;
47      }
48
49      private boolean remoteFileExist(){
50        boolean result=false;
51        bufferOutputTCP.writeInt(ID);
52        bufferOutputTCP.writeUTF("remoteFileExist");
53        bufferOutputTCP.writeUTF(model);
54        bufferOutputTCP.flush();
55        result=bufferInputTCP.readBoolean();
56        return result;
57      }
58
59      private void createModel(long fileLength){
60        byte modelBytes[];
61        FileInputStream fstream = new FileInputStream(new File(model));
62        DataInputStream in = new DataInputStream(fstream);
63        modelBytes= new byte[(int)fileLength];
64        in.readFully(modelBytes);
65        in.close();
66        //Create model in remote server
67        model=importModel(model,modelBytes);
68        ...
69      }
70      ...
71    }
```

Listing 3.26: `RSimulinkExternalApp` class: Some of the methods implemented.

The constructor is similar to the class `RMatlabExternalApp`. However, here the name of the Simulink model is required from the string `config`. The method `connect` also shows differences. The main one is the necessity to check for the existence of the Simulink model on the client side or on the remote side. If the model is located on the client side then the model is sent to the remote server. The upload of the model is executed by the methods `createModel` and `importModel`.

Probably the most interesting method in the class `RSimulinkExternalApp` is the implementation of the asynchronous step named `stepAS` as in the `RMatlabExternalApp`. This method works slightly differently here (see Listing 3.27). This is because there is no synchronization between client and remote application, and the Simulink model can be executed at a different time than the client application shows. Hence, when the end

user interacts with the simulation, the time at which the Simulink model should be reset is probably different from the time of the Simulink model. To solve this problem, the Simulink model has to be stopped and restarted again at the time given by the client application. Note that this could produce some problems, since some Simulink models execute initialization functions when they are started. So, the author of the interactive simulation should consider how to fix this. Fortunately, these cases are not common in most simulations, and it is always possible to execute the simulation in synchronous mode.

In Listing 3.27 the `stepAS` method is described. Note that it is similar to the version for `RMatlabExternalApp`, but the set of `eval` indicates that there are some differences as explained before. This set of evaluations just initiate the state of the Simulink model when it will be stopped. So, the initial state of the model when it is restarted will be the last state that the model had before being stopped.

```
1  protected void stepAS (double dt) {
2    if (isSynchronized){
3      haltUpdate(true);
4      setValues();
5      //Reset all states of the Simulink model
6      eval("xFinal=[]",false);
7      eval ("set_param ('"+model+"', 'LoadInitialState','on')",false);
8      eval ("set_param ('"+model+"', 'InitialState','xFinal')",false);
9      eval ("set_param ('"+model+"', 'SaveFinalState','on')",false);
10     eval ("set_param ('"+model+"', 'FinalStateName','xFinal')",false);
11         //Send information to the server about the Step required
12     bufferOutputTCP.writeInt (ID);
13     bufferOutputTCP.writeUTF ("stepSimulinkAS");
14     ...
15     bufferOutputTCP.flush();
16     isSynchronized=true;
17   }
18   //Get Values
19   getValues();
20 }
```

Listing 3.27: `RSimulinkExternalApp` class: Asynchronous step.

Listing 3.28 shows the piece of code required to attend the asynchronous step in the JIM server. This function first tries to get the name and class of the variables to be sent back to the client application. Note that the MATLAB structure `vars`(created when the model is modified) is used to get from the field `fromto` the output variables of the Simulink model.

```
1  if (RemoteFunction.equalsIgnoreCase("stepSimulinkAS")) {
2    ...
3    if (theFirstTime) {
4      theFirstTime = false;
5      //Get from MATLAB workspace the variables to send back
6      matlab.engEvalString(id,"var2back=vars.name((strcmp(vars.fromto,'out')))");
7      ...
```

```
8       //Get the Output types
9       outVars = getVars("varComma");
10      classVar = getClass(outVars)
11      ...
12    }
13    //Get information from the client about the Step required
14    model=(String)bufferInput.readUTF();
15    //Set Start Time
16    matlab.engEvalString(id,"set_param ('"+model+"', 'StartTime','time')");
17    ...
18    //Stop and start the model
19    matlab.engEvalString(id,"set_param('"+model+"','SimulationCommand','stop')");
20    matlab.engEvalString(id,"set_param('"+model+"','SimulationCommand','start')");
21    ...
22    do{
23      for (int i = 0; i < steps; i++) stepModel();
24      //Get values from MATLAB
25      for (int i = 0; i < outVars.length; i++) {
26        switch (classVar[i][0]) {
27          case 1: //Get String
28                          ...
29          case 2: //Get doubles
30            //double
31            if ( (classVar[i][1] * classVar[i][2]) <= 1) {
32              double value = matlab.engGetScalar(id,outVars[i].trim());
33              bufferOutput.writeDouble(value);
34            }
35            //double Array
36                          ...
37            //double Matrix
38        }
39      }
40      bufferCount++;
41      if (buffercount >= PACKAGE_SIZE){
42        bufferOutput.flush();
43        buffercount = 0;
44      }
45    }while ((bufferInput.available()==0) && jimMonitor.stateServer());
46    bufferOutput.flush();
47    buffercount = 0;
48    synchronize = true;
49  }
```

Listing 3.28: JIM server: code to support the asynchronous step for Simulink models.

After the name and class of the output variables are obtained, the `StartTime` parameter, which represents the initial time of the simulation, is set to the value given by the MATLAB variable `time`. This variable has been previously modified (by the method `setValues`) to the value of the client variable. The model is then stopped and started using the parameter `SimulationCommand`.

Once the model is started, a do-while cycle is executed. This cycle ends when data arrives from the client to the input buffer or when the JIM administrator stops the server. Within the cycle, the model is stepped as many time as the variable `step` indicates. Then, depending on the values of the variables `outVars` and `classVar`, the server sends back to the client the corresponding values of the output variables. Note that these output values are accumulated in a package, which is actually sent to the client side when the value of `buffercount` is greater than `PACKAGE_SIZE`.

The rest of the `RSimulinkExternalApp` class is either already implemented by the `RMatlabExternalApp` or very close to methods described in the `SimulinkExternalApp` class. In addition, other options implemented in the `SimulinkExternalApp` class, such as those needed to speed up the simulation, can be used. The next section includes further discussion about the use of the `RSimulinkExternalApp` Java class.

Using the class `RSimulinkExternalApp`

The use of the `RSimulinkExternalApp` class is quite similar to the previous class, `SimulinkExternalApp`. In fact, any example developed with the `SimulinkExternalApp` class can be easily transformed to perform a remote simulation.

For instance, the example shown in Listing 3.20 can be modified, in order to use the class `RSimulinkExternalApp`, by simply changing the following line:

```
SimulinkExternalApp externalApp =
  new SimulinkExternalApp("fsmk.mdl");
```

to the following line:

```
RSimulinkExternalApp externalApp =
  new RSimulinkExternalApp(<matlab:ipserver:portserver>"fsmk.mdl");
```

where the strings `ipserver` and `portserver` are the IP address and the Port number of the JIM server.

The execution of this new example produces exactly the same output for the evaluation of the function as before. In order to use the asynchronous link with the Simulink model, the constructor of the class should be initiated as follows:

```
RSimulinkExternalApp externalApp =
  new RSimulinkExternalApp(<matlabas:ipserver:portserver>"fsmk.mdl");
```

The same modifications can be introduced in the example of Listing 3.21 in order to have a synchronous or asynchronous link with the remote version of the bouncing ball.

3.5.3 The package JIMC

The package JIMC is a freely available open source Java library (Department of Computer Science and Automatic Control, UNED 2010*a*). Authors who want to implement their own interactive simulation in Java using this library, can find and download it with some examples at:

`http://lab.dia.uned.es/rmatlab`.

In this web location authors can also find the JIM server.

The library contains the four Java classes described in this chapter to control the software MATLAB/Simulink. Table 3.4 summarizes the classes implemented by the package JIMC.

Table 3.4: The available classes in the Java package JIMC.

Class	Description
`MatlabExternalApp`	This class allows users to manipulate MATLAB from Java. The class implements the Java interface `ExternalApp`.
`SimulinkExternalApp`	The class extends the class `MatlabExternalApp` and adds special methods to manipulate MATLAB/Simulink from Java.
`RMatlabExternalApp`	A class to manipulate a remote MATLAB from Java. It requires to start the JIM server where the remote MATLAB is installed. The class implements the Java interface `ExternalApp`.
`RSimulinkExternalApp`	This class extends the class `RMatlabExternalApp` to manipulate remote Simulink models from Java. It requires to start the JIM server where the remote MATLAB is installed.

In order to use the package, it must be imported by adding the following line in the code of a Java application:

```
import jimc.*;
```

The described classes can then be used in the same way as in the previous examples.

3.6 Interfacing other engineering software with Java

Similar to the cases described before, other implementations with different engineering software can be performed in order to support the communication protocol of the interoperate approach. The first and main requirement is the possibility to call the desired engineering software from Java. Fortunately, this feature is present nowadays in most engineering software, which is commonly named External Interface.

Following the implementations developed for MATLAB and Simulink, this section discusses how to create a Java class that implements the `ExternalApp` interface for two engineering software: Scilab and Sysquake. Both implementations describe only the local connection, but they could also be developed for remote operations as before.

3.6.1 Scilab

Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications, see Figure 3.21. Scilab was developed in 1990 by INRIA and the École Nationale des Ponts et Chaussées(ENPC). Since the creation of the Scilab consortium in May 2003, it has been developed and maintained by the INRIA. The software is distributed freely along with the source code via the Internet (Scilab Consortium 2010).

Scilab is a high-level and numerically oriented programming language. It provides an interpreted programming environment, using matrices as main data type. The software can be used for signal processing, statistical analysis, image enhancement, fluid dynamics simulations, and numerical optimization. The syntax of the language is quite close to the language provided by MATLAB. For this reason, many of the functionalities implemented in MATLAB can be found in Scilab as well. So, similarly to MATLAB, this software expands its capability by using specific toolboxes called Modules.

The package also includes a module called Scicos for modelling and simulation of explicit and implicit dynamical systems, including both continuous and discrete subsystems. This toolbox (written in Fortran, C, and Scilab language) provides many functionalities available in Simulink.

A Java class to manipulate Scilab

Since version 4.0, a Java interface was incorporated to Scilab. This interface allows calling Scilab's computational engine from Java programs. The link works similarly to the functionality provided by JMatLink to manipulate MATLAB, i.e., there is a Java package named **javasci.jar** which calls a dynamic or shared library (**Javasci.dll**) by using the Java Native Interface to control Scilab.

The methods implemented in the Javasci package are a little less sophisticated than the methods provided by JMatLink. There are only a few methods to execute Scilab commands and to get values of variables previously created by using the package. However, it is not difficult to develop the functionality required by the Interoperate Approach. For example, Listing 3.29 shows some methods implemented (`getDoubleArray` and `setValue`) to support the low-level protocol. Other methods required to support the high-level protocol are similar to the corresponding implementation described for

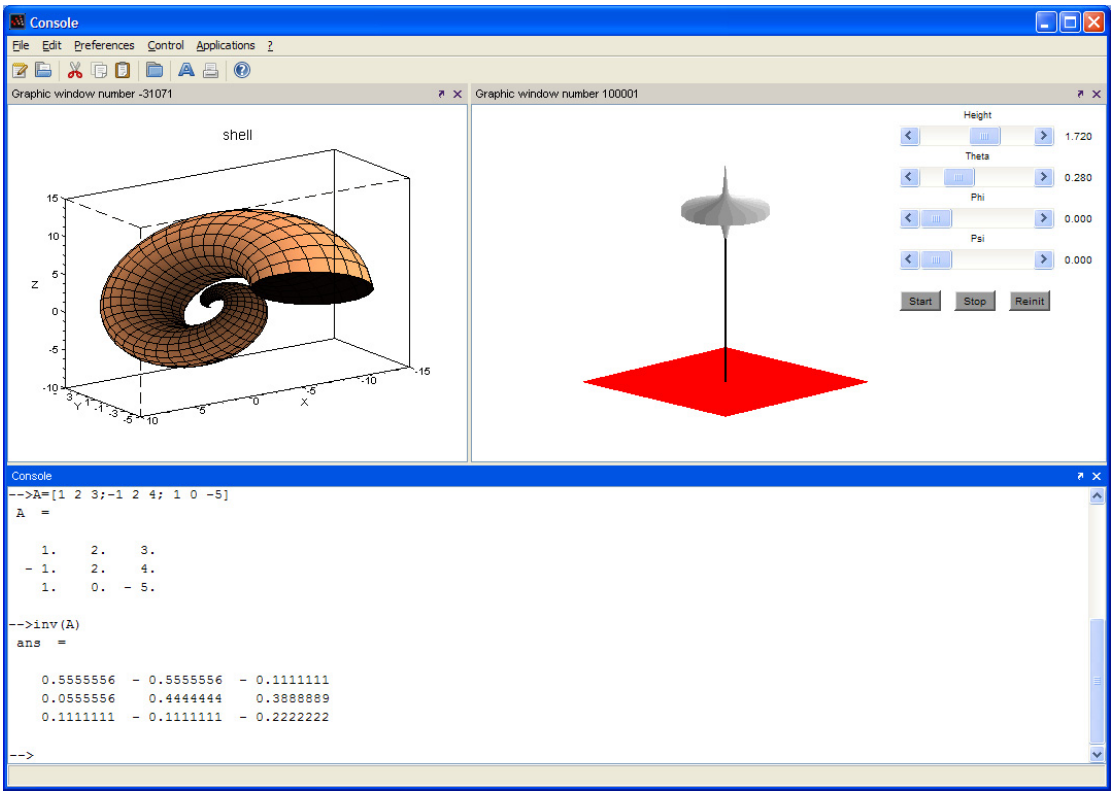


Figure 3.21: Scilab software.

the Java class `MatlabExternalApp`.

```
1
2  public class ScilabExternalApp implements ExternalApp {
3    ...
4    public double[] getDoubleArray (String varName) {
5      if (!scilabConnected) return new double[]{0};
6      SciDoubleArray sizeScilab = new SciDoubleArray("EjsSciLength",1,2);
7      Scilab.Exec("EjsSciLength=size("+varName+");");
8      double[] size = sizeScilab.getData();
9      SciDoubleArray var=new SciDoubleArray("___"+varName,(int) size[1],1);
10     Scilab.Exec("___"+varName+"="+varName+";");
11     double[] value=var.getData();
12     Scilab.Exec("clear ___"+varName+" EjsSciLength"+";");
13     return value;
14   }
15
16   public void setValue (String varName, double[] value) {
17     if (!scilabConnected) return;
18     new SciDoubleArray(varName,1,value.length,value);
19   }
20   ...
21 }
```

Listing 3.29: A Java class that implements the communication protocol for Scilab.

Using the Java class to manipulate Scilab.

The use of the Java class that implements the communication protocol is similar to the previous examples shown for MATLAB and Simulink. For instance, Listings 2.5 and 2.8 of Chapter 2 can be easily modified in order to use Scilab as the external application.

Thus, the following line:

```
ExternalApp externalApp = new MyExternalApp();
```

has to be replaced by:

```
ExternalApp externalApp = new ScilabExternalApp();
```

But, since the Scilab language is not exactly the same as MATLAB, the function to be evaluated has to be modified from:

```
y=sin(2*pi*f*t)*cos(t)
```

to

```
y=sin(2*%pi*f*t)*cos(t)
```

Other, more elaborated examples, such as the one shown in Listing 3.12 for MATLAB, can be also easily transformed to be used with Scilab.

3.6.2 Sysquake

Sysquake is an innovative, powerful, and flexible software for understanding systems, solving problems, and designing products (Calerga 2010). It is used in teaching, research, and engineering. Sysquake is a numerical computing environment based on a programming language almost compatible with MATLAB.

It also offers facilities for interactive graphics, which give insights into the problems being analyzed. The interactivity provided in the graphics is quite close to that required for a good and quick understanding of the relations between different variables. For example, on the right-side of the Figure 3.22, there is an interactive graph which shows the response of a linear system controlled by a digital controller. Thus, when users drag one of the closed-loop poles with the mouse (shown as crosses on the upper left of the graphic) a new controller is computed and all the figures are updated nearly instantaneously. This way, users see how the system responds to the manipulations and can observe, for instance, how the frequency of the step response is related to the position of the poles. This interactivity aids in understanding how quantities are related to each other and designing better controllers extremely quickly.

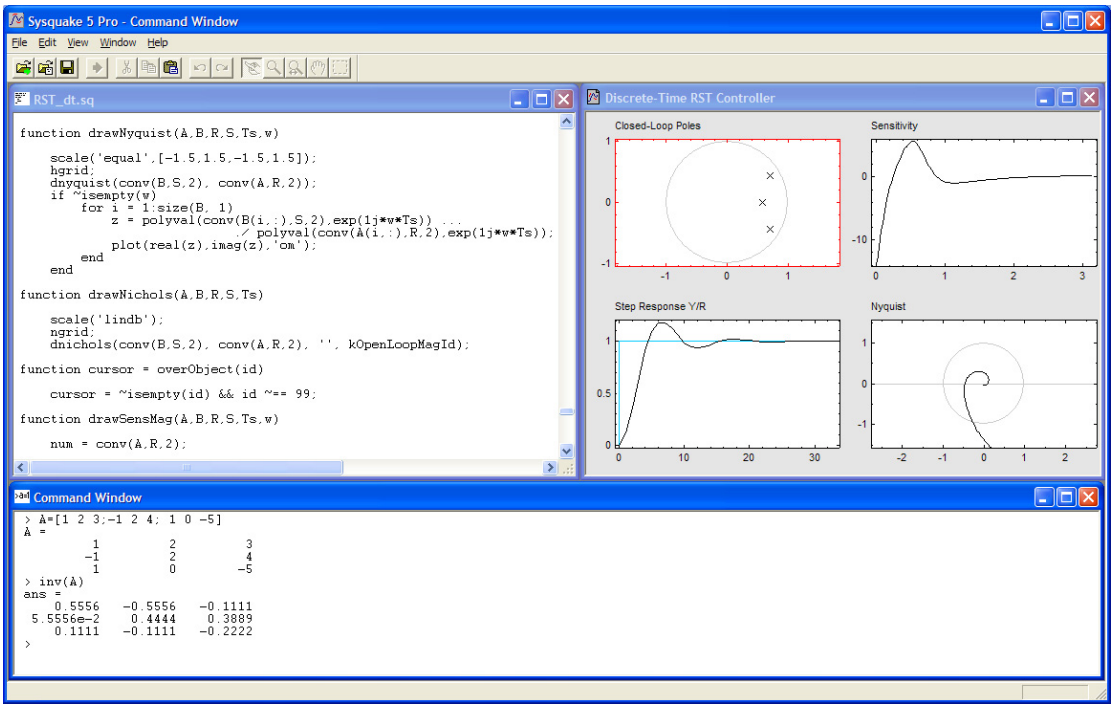


Figure 3.22: Sysquake software.

A Java class to manipulate Sysquake

SysquakeLink is a Java package to communicate with Sysquake 3.5 and later from other applications. It relies on OLE Automation on Windows and on the XML-RPC server of Sysquake on unix platforms (Mac OS X and Linux). Similar to JMatLink or Javasci packages, SysquakeLink uses the Java Native Interface to provide a link between Java programs and Sysquake.

SysquakeLink is made of a set of cross-platform Java classes and architecture dependent native methods. Java classes are written in such a way that native methods are loaded from the shared libraries (.so or .dll files).

The methods implemented in the SysquakeLink package are similar to those described for the JMatLink and Javasci packages. As in previous cases, it is not complicated to implement the communication protocol of the interoperate approach. Listing 3.30 describes the implementations of two methods of the low level protocol for this class: `getDoubleArray` and `setValue`. Other methods to implement the high level protocol are similar to the implementations shown in the case of the Java class `MatlabExternalApp`.

```
1
2  public class SysquakeExternalApp implements ExternalApp,
3               com.calerga.sysquake.SQLinkVariableListener{
4    ...
5    public double[] getDoubleArray (String variable) {
6      Object obj = null;
7      Integer n = varTable.get(variable);
8      try {
9       if (n != null) obj = SysquakeLink.variableValue(sqID,variable);
10      else           obj = SysquakeLink.lmeVariableValue(variable);
11      if (obj instanceof double[][]) return ((double[][])obj)[0];
12      return (double[]) obj;
13     }catch (Exception e) {e.printStackTrace();}
14     return null;
15   }
16
17   public void setValue (String variable, double[] value){
18     Integer n = varTable.get(variable);
19     if (n != null)
20       try {
21         SysquakeLink.setVariableValue(sqID,variable,value);
22       }catch (Exception e) {e.printStackTrace();}
23     else {
24      StringBuffer cmd = new StringBuffer(variable);
25      cmd.append(" = [ ");
26      for (int i=0; i<_value.length; i++) {
27       if (i>0) cmd.append(",");
28       cmd.append(Double.toString(value[i]));
29      }
30      cmd.append("];");
31      try {
32        SysquakeLink.execute(cmd.toString());
33      }catch (Exception e) {e.printStackTrace();}
34     }
35   }
36   ...
37 }
```

Listing 3.30: A Java class that implements the communication protocol for Sysquake.

Using the Java class to manipulate Sysquake.

The use of the Java class that implements the communication protocol is similar to the previous examples shown for MATLAB, Simulink, and Scilab. Thus, Listings 2.5 and 2.8 of Chapter 2 can be again easily modified to use Sysquake as the external application. So, the following line:

```
MyExternalApp externalApp = new MyExternalApp();
```

is replaced by:

```
ExternalApp externalApp = new SysquakeExternalApp();
```

Unlike the example shown for Scilab, the command does not need to be modified in order to use Sysquake.

Other, more elaborate examples, such as the one shown in Listing 3.12 for MATLAB, can be also easily transformed to be used with Sysquake.

3.7 Conclusions

The chapter presents the implementation of Java classes for standard engineering software tools according to the communication protocol of the interoperate approach presented in Chapter 2. Some examples of use of each Java class are shown. The implementation requires that tools provide a way to interface with it. Fortunately, the existence of such interfaces is a common feature of standard engineering software tools.

The first external application implemented is the `MatlabExternalApp` class, which allows the interoperation of MATLAB from a Java program. The `MatlabExternalApp` class uses the open source library JMatLink mainly to implement the low-level protocol. Other methods, required by the high-level protocol, are also implemented following the requirements of the communication protocol.

Since Simulink models can be controlled from MATLAB, is possible to use the `MatlabExternalApp` class to run interactive simulations using Simulink models. However, this approach demands many changes in the original Simulink model in order to successfully execute the interactive simulations. For this reason, in order to generate automatically all the changes needed by the Simulink model, the Java class `SimulinkExternalApp` was implemented. The `SimulinkExternalApp` class extends the `MatlabExternalApp` class by overriding mainly the methods to link variables and to step the simulation.

A remote interoperation was described also for both MATLAB and Simulink simulations. The Java classes `RMatlabExternalApp` and `RSimulinkExternalApp` implement the client side of the remote operation, whereas the server side is supported by the**JIM server**. The two type of remote links, synchronous and asynchronous, are implemented as well. The `synchronize` method is introduced here in order to treat correctly the integrator blocks of Simulink models and to synchronize asynchronous remote operations.

The four Java classes to manipulate MATLAB and Simulink, local and remotely, are packaged into the freely available Java library JIMC. This library and the JIM server can be found at:

`http://lab.dia.uned.es/rmatlab`

The chapter ends describing, similarly to MATLAB and Simulink, how to implement the communication protocol in other external applications such as Scilab and Sysquake.

After the implementation of the communication protocol has been described, the next chapter focuses on facilitating the creation of interactive human interfaces.

Chapter 4

Interactive Applications Using Engineering Software

Once the implementation of the Java interface `ExternalApp` has been described for various Engineering Software, the next step is to build interactive applications with pedagogical purposes. This can be a very hard task for teachers, who are not used to programming rich graphical user interfaces. This is why this chapter introduces the Easy Java Simulations authoring software tool.

Easy Java Simulations helps instructors to create complex Java applications, with a high level of user interaction and sophisticated computer graphics, which, however, does not require expert knowledge in Java programming.

The chapter begins with a brief introduction on the use of Easy Java Simulations. It describes in detail how to integrate the package JIMC in the applications created with Easy Java Simulations. Although this process does not present difficulties, a built-in feature was developed in Easy Java Simulations to manipulate not only MATLAB/Simulink, but also other engineering applications such as Scilab or Sysquake. The facilities provided by this built-in feature will ease, even more, the creation of interactive applications that use engineering software.

The final section of the chapter describes a real-life example of the integration between Easy Java Simulations and MATLAB. The application developed, a networked control laboratory, was used in a basic engineering control course in Ghent University, and received positive feedback from the students.

4.1 Easy Java Simulations: EJS

Easy Java Simulations(EJS) is an open source (and therefore completely free) software tool designed to create simulations in Java with high-level graphical capabilities and with an increased degree of interactivity (Esquembre 2004, 2005, 2010). The tool provides its own mechanism for describing models of scientific and control engineering phenomena, and, as such, can be used to create virtual laboratories on its own. Figure 4.1 shows the EJS user interface.

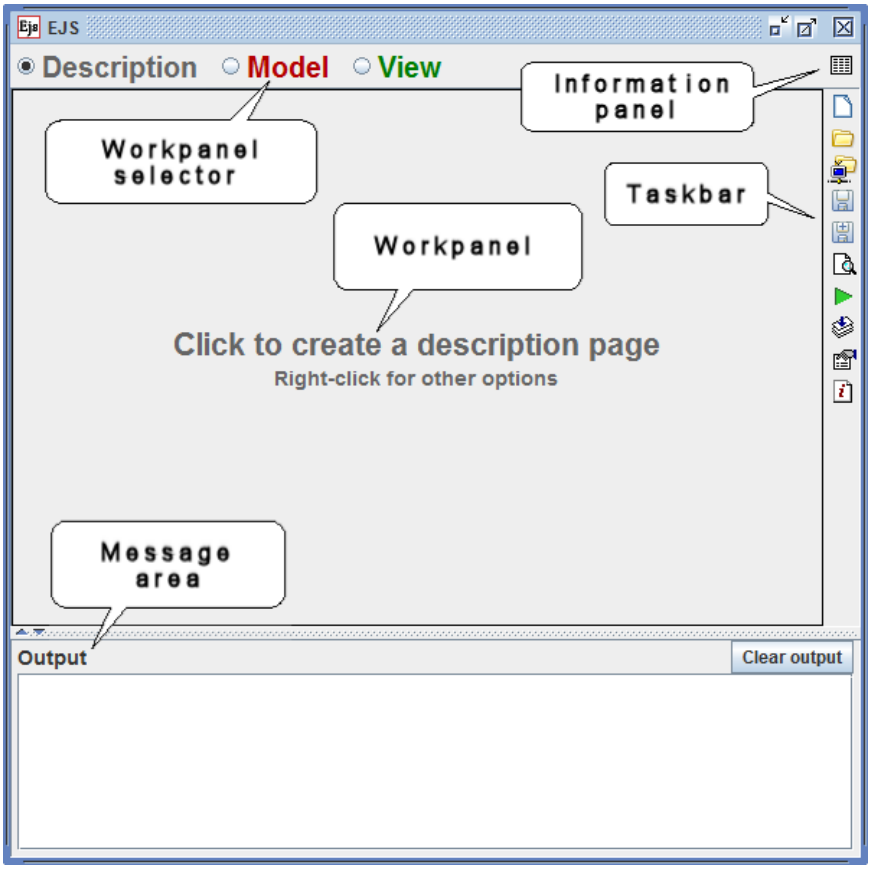


Figure 4.1: The graphical user interface of EJS.

EJS is different from most other authoring tools in that it is not designed to make life easier for professional programmers, but has been conceived for science students and teachers. That is, for people who are more interested in the content of the simulation, the simulated phenomenon itself, and much less in the technical aspects needed to build the simulation.

The tool structures a simulation in two main panels, the **Model** and the **View**. Apart from the Model and the View, there is also an introductory part, named **Description**, to describe (using HTML files) the system to be simulated.

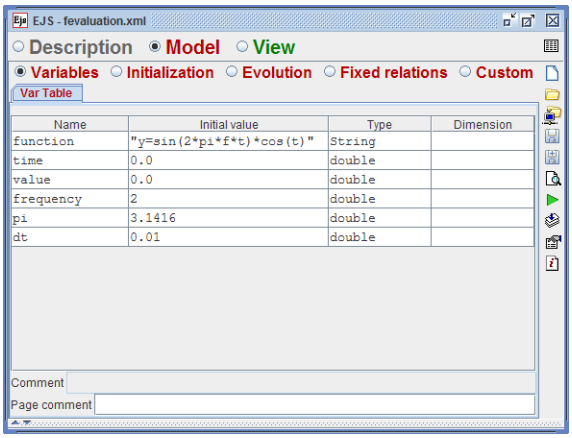


Figure 4.2: Subpanel Variables of EJS.

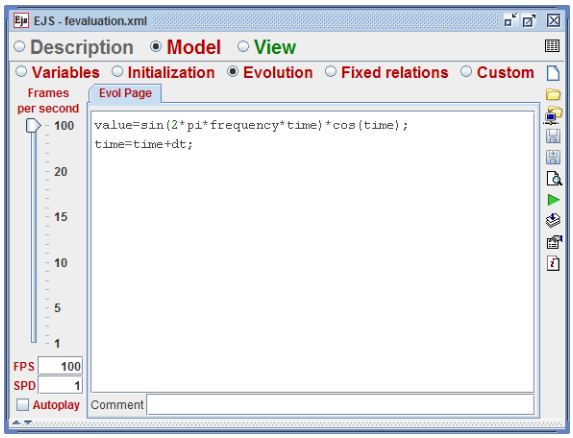


Figure 4.3: Subpanel Evolution of EJS.

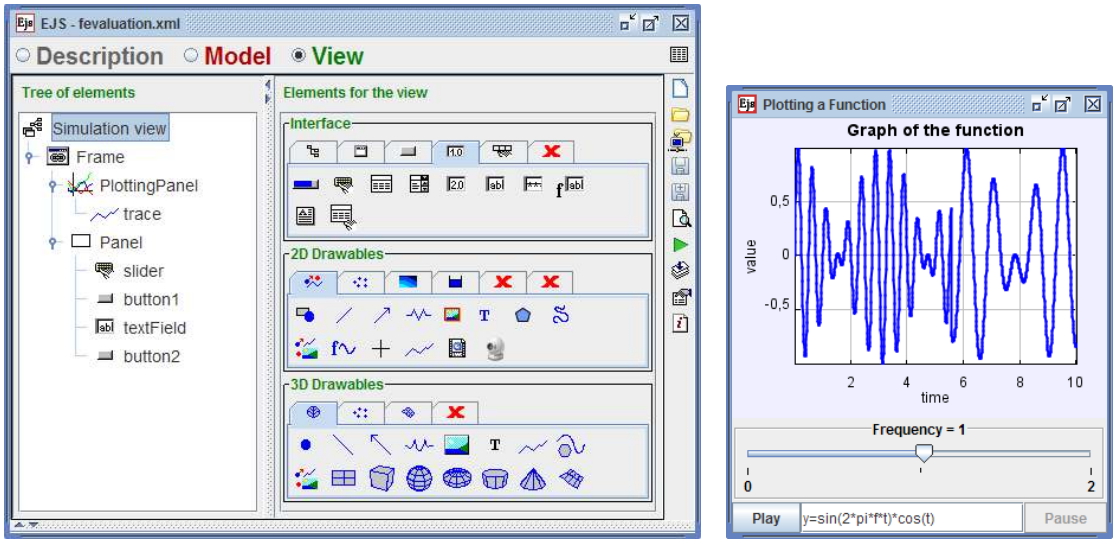
The Model describes the simulated system by means of variables (both state variables and parameters), that completely characterize the system, and of computer algorithms that state how the system evolves in time and how it responds to user interaction. Authors need to declare the variables using a simple table, and write the Java code needed to specify the algorithms. EJS offers specialized help to solve models based on ordinary differential equations (ODEs) by providing an editor to write these equations and automatically generating the code required using the most popular solvers.

The Model is divided in five subpanels: `Variables`, `Initialization`, `Evolution`, `Fixed relations` and `Custom`. In the Variables subpanel the global variables of the simulation are declared. The Initialization subpanel allows authors to execute code of initialization before stepping the simulation. In the Evolution subpanel authors can input two type of descriptions: pure Java code or ordinary differential equations by using the ODEs editor. Both types of descriptions are evaluated continuously while the simulation is performed. The Fixed relations subpanel provides an additional way to execute Java code when the user interacts with the view while the simulation is paused. The Custom subpanel can be used by authors to implement their own Java methods. Figure 4.2 and Figure 4.3 show the Variables and Evolution subpanels respectively.

The View provides the visualization of the simulated system, either in a realistic form or using one or several data graphs, and the user interface elements required for user interaction. These **view elements** can be chosen from a set of predefined, ready-to-use components, to build a tree-like structure in a kind of block construction game for the view. There are elements of several types. Each type specializes in a given visualization or interaction task, but can also be customized using the so-called **properties**, a set

of internal values that modify the aspect and behaviour of the element on the screen. This way, the job of the author when building the view consists in choosing the right elements from those offered and in customizing them for the interaction desired for the simulation. Figure 4.4a shows the View used to build the graphical user interface shown in Figure 4.4b.

Both, model and view need to be interconnected. Any change in the model state must be immediately reflected by the view in order to keep a dynamic, on-the-fly visualization of the system. In turn, any interaction of the user with the view must immediately affect the model so that the desired interactivity is achieved. This communication is based on connecting model variables and view elements properties. This connection is very easily established by typing, in the table of properties of view elements, the names of the model variables to be connected to the properties. Once the model and the view have been created and the required connections established, EJS creates the ready-to run simulation at a single mouse click, taking care of a good number of technical issues that thus becomes completely transparent to the author. The result is an independent, high performance, interactive simulation which can either be run as a stand-alone Java program, or be embedded as an applet in an HTML page. More description about using Easy Java Simulation, some examples, and the software can be obtained from *http://www.um.es/fem/Ejs/*.



a) b)

Figure 4.4: a) The View panel of Easy Java Simulations. b) A graphical user interface created by the View shown in a).

In order to give a brief introduction about how EJS can be used to create interactive

simulations, two examples are presented in next sections.

4.1.1 A first example with EJS: evaluation of a function

Consider again the problem of evaluating a function described in Listings 2.5 and 2.8 of Chapter 2. This simulation can be easily implemented in EJS.

The first step is to declare the variables of the simulation. In this case the following six are needed: `function`, `time`, `value`, `frequency`, `pi`, and `dt`.

All the variables are of type double, except the variable `function` which is of type String. The declaration of these variables is done by using the Variables subpanel of EJS. After the declaration, the section should look like the example in Figure 4.2.

The computation of the function can be declared in the Evolution subpanel of EJS. This computation is executed continuously while the simulation is running. After each evaluation of the function, the time has to be incremented in order to perform the simulation. The code required to evaluate the function and to increment the time is shown in Figure 4.3. The trigonometric functions for the sine (`sin`) and the cosine(`cos`) are obtained from the Java package Math. This package is imported by the following statement[1]:

```
static java.lang.Math.*;
```

The statement has to be entered in the parameter `Imports` of the Information Panel of EJS as Figure 4.5 shows.

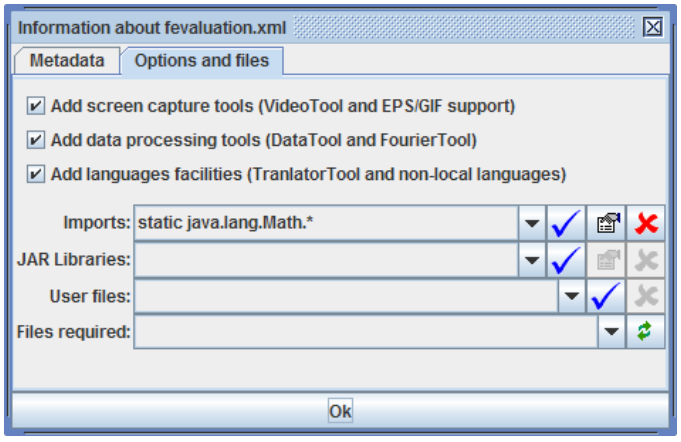


Figure 4.5: Declaring the Math Java package.

After the model of the simulation is defined, the view is built using the visual elements

[1]In EJS the import of the Math Java package is not needed. This package is imported in EJS by default. This is done here only for illustrative purposes.

provided by EJS. The user interface developed is shown in Figure 4.4b and the visual elements used for it are shown in Figure 4.4a. Note that the tree-like structure gives shape to the view. Two types of windows can be added to the simulation view, the root of the tree: `Frames` or `Dialogs`. The first type is used normally as the main window of the simulation. The second type can be selected to show a subordinate window. In this simulation, a frame has been selected.

Two panels have been added to the main window: a `PlottingPanel` and a basic `Panel`. The first one is used normally to display a panel with axis, where visual elements from the set of `2D Drawables` can be added to plot a graph. For instance, in this simulation a `trace` is used to display the value of the sinusoidal function versus time. The basic `Panel` element is simply used to organize suitably the position of the other visual elements in the user interface.

The `Panel` has four children elements: `slider`, `button1`, `textField`, and `button2`. The first component is a slider used to modify the frequency of the sinusoidal function. The second and fourth elements are two push buttons to play or pause the simulation. The `textField` shows the function being evaluated.

In order to connect the model and the view of this simulation, the properties of the visual elements are customized as Figure 4.6 shows.

In Figure 4.6a the properties of the button1 `Text` and `Action` are modified to show the text *Play* and to play the simulation by using the predefined method `_play()`. The action of the button is triggered when users push the button. In the case of button2, the text and the action are *Pause* and the predefined method `_pause()` respectively.

In Figure 4.6b the property `Variable` is used to display the value of the `function` String variable. Note that here any action is executed when the user interacts with this element.

Figure 4.6c shows the properties for the trace. This element plots a line, where each new point is given by the properties `Input X` and `Input Y`. These properties are updated with the values of time and value respectively as the simulation is running.

The frequency of the sinusoidal function is controlled by the slider. The connection between this visual element and the variable frequency is defined in the `Variable` property as Figure 4.6d shows.

This simple example introduces the use of EJS to build interactive simulations. More

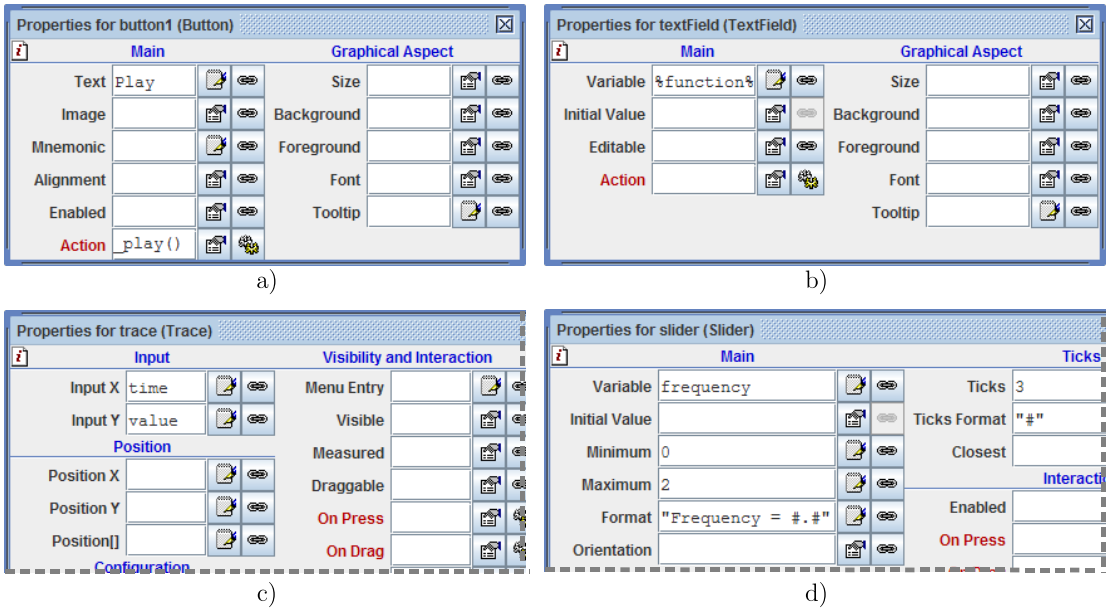


Figure 4.6: Properties for some visual elements. a) button1 b) textField c) trace d) slider.

elaborate examples can be found in (Esquembre 2004, 2005, Dormido & Esquembre 2003, Dormido et al. 2004, Sánchez et al. 2010). However, the creation of the example can be compared to the simulation in Chapter 3 shown in Figure 3.3. Note that the amount of code required by the EJS simulation is much less than that needed by the example of Chapter 3 as described in Listing 3.12. This is a consequence of the simplified way that EJS provides to build the interactive user interfaces.

4.1.2 A second example with EJS: manipulating ODEs

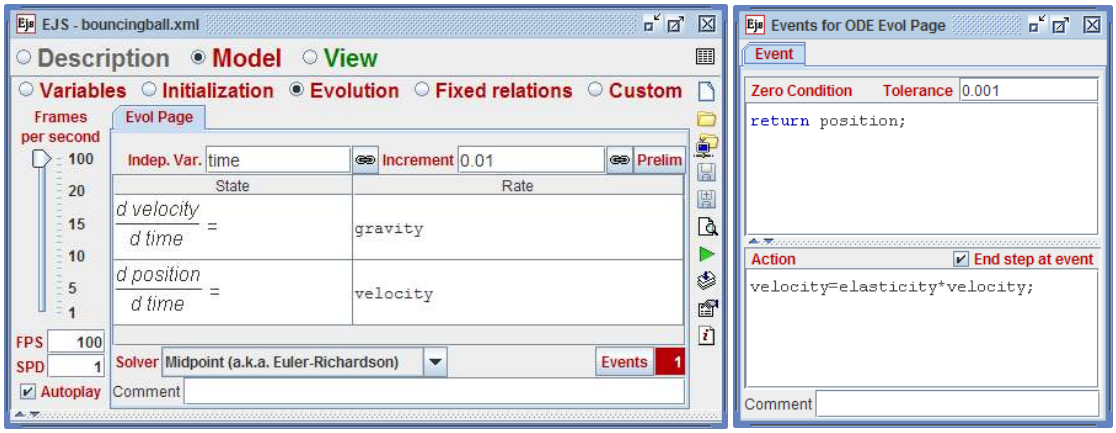
EJS can be also used to simulate directly a model described by ordinary differential equations. For example, consider again the model of a bouncing ball. In this case, the model consists of two first-order ODEs:

$$\begin{aligned} \frac{dv}{dt} &= g \\ \frac{dp}{dt} &= v \end{aligned} \tag{4.1}$$

where t is the time, g the gravity, v is the vertical velocity, and p is the vertical position of the ball.

Assuming that variables have already been declared in the `Variables` subpanel, the model can then be easily described in EJS by using the special editor of ODEs. The

editor, which is located in the model part of the simulation, allows authors to enter the required ordinary differential equations. Figure 4.7a shows the model of the simulation of the bouncing ball in EJS. Note that, apart from the equations, authors have to indicate: the independent variable (*time*, in this case), the size of the integration step (0.01), and the solver algorithm to perform the integration of the equations.



a) b)

Figure 4.7: The model of the bouncing ball. a)The ODEs of the system b)The event when the ball hits the ground.

The model of the bouncing ball needs also to account for the event triggered when the ball hits the ground. In EJS, events can be added easily by using the event editor. This editor is accessed by clicking on the `Events` button, which is located next to the solver list (see Figure 4.7a). The event is edited introducing two parts: the `Zero Condition` and `Action`.

The `Zero Condition` is used to detect the exact moment when the event occurs, i.e. when the value returned by the `Zero Condition` passes from positive to negative.

The `Action` is used to execute the required actions when the event is triggered.

The event for the bouncing ball is shown in Figure 4.7b. Note that the event in this case occurs when the ball's vertical position switches from a positive to a negative value (i.e., when the ball hits the ground). Once the event is triggered, the velocity of the ball is recalculated by multiplying its previous value by the elasticity of the ball[2].

Once the model description is finished, the view of the simulation is created in EJS as displayed by Figure 4.8. Here, two new visual elements are introduced: A `DrawingPanel` and a `Particle` element named `ball`. The first element is a basic container for 2D drawing. This element can contain any element of the group `2D drawables`. The `ball`

[2]Note that here, for convenience, the elasticity is considered negative.

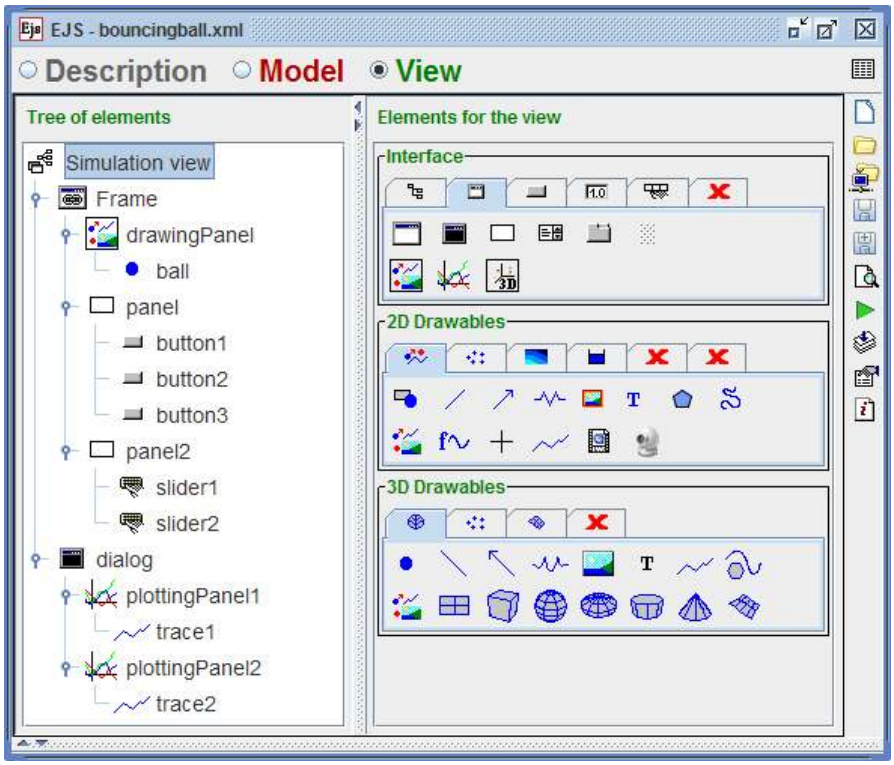


Figure 4.8: The design of the view of the bouncing ball in EJS.

is an element used to display a circular shape which looks like a ball. The properties of the ball are described in Figure 4.9. The main parameters of the particle (`Pos X` and `Pos Y`) define the position of the ball. Other three properties (`On Press`, `On Drag` and `On Release`) are used to process any user interaction with the ball. Thus, when the user clicks on the ball to drag it to another position, the simulation is paused. Once the user releases the ball, the simulation is restarted but with its velocity reset to zero.

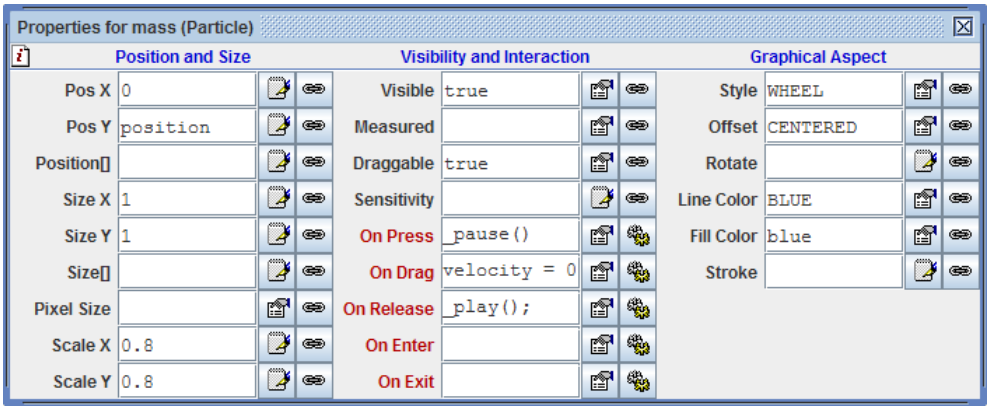
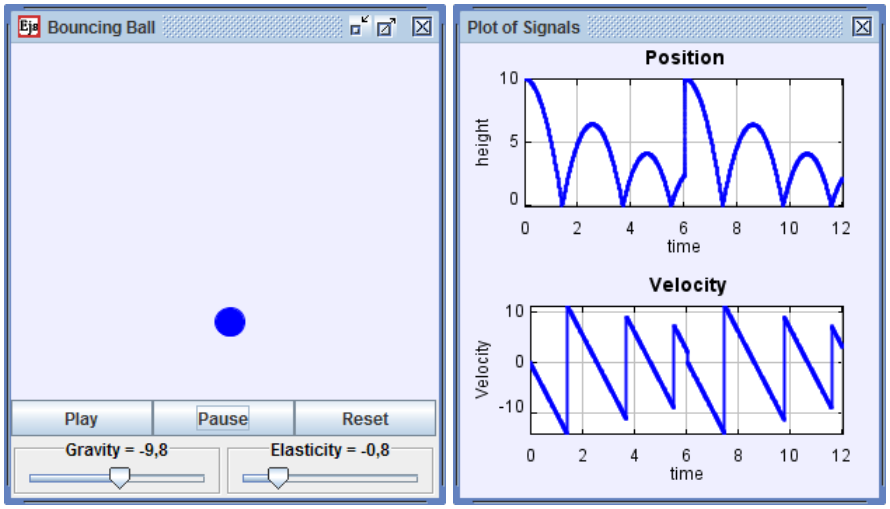


Figure 4.9: The properties of the visual element `ball`.

Apart from the `DrawingPanel` and the `Particle`, there are also other visual elements to add interactivity to the simulation. The two sliders (`slider1` and `slider2`) allow

end users to modify the gravity and the elasticity parameters respectively. The three buttons (`button1`, `button2`, and `button3`) allow end users to play, pause, and reset the simulation. Note also that, similarly to the previous EJS example, two plotting panels and two traces are used to draw the position and velocity of the ball.

The result of the view created in Figure 4.8 is shown in Figure 4.10. Observing the graphs of velocity and position (Figure 4.10b), it can be inferred that the ball was dragged and released by the end user from 2.5m to 10m at time = 6s.



a) b)

Figure 4.10: User interface of the simulation of the bouncing ball.

4.2 Using the JIMC package from EJS

As described in Chapter 3, authors can build interactive simulations using Java and MATLAB/Simulink software. The link between MATLAB and Java is provided by the Java class `MatlabExternalApp` of the JIMC package. Since it is possible to use from EJS any Java package (as the Math package of the previous section), authors can select the package JIMC in order to create simulations with MATLAB in a similar way to that shown in Chapter 3.

In order to use the JIMC package, authors need first to import this library. This is done by using the Information Panel of EJS, and adding to the `Imports` field the following statement:

```
jimc.*;
```

The location of the package in the hard disk needs also to be indicated in the `JAR`

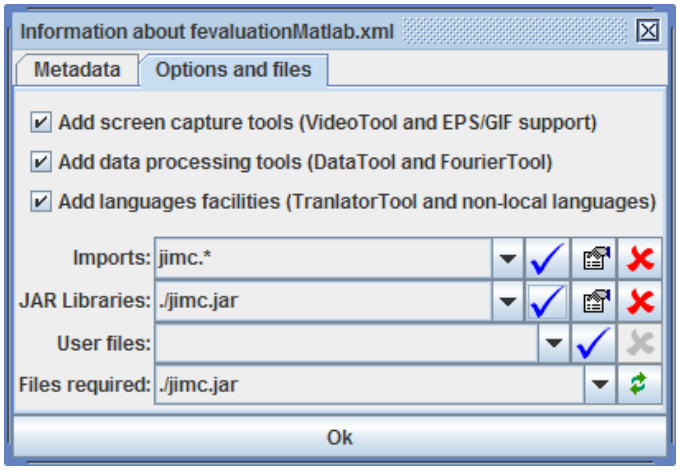


Figure 4.11: Declaring the package JIMC.jar.

`libraries` field of the Information Panel of EJS. This is needed because, unlike the Math package used before, JIMC is a user-defined Java package. Figure 4.11 shows the Information Panel of the simulation after JIMC has been defined.

After importing the JIMC package, authors can use any of the four classes that the library contains.

4.2.1 Using the `MatlabExternalApp` Java class from EJS

The first example with EJS shown in Subsection 4.1.1 can be revisited in order to obtain a similar simulation but now using MATLAB to evaluate the function.

After importing the JIMC package, the variables have to be declared. In this example, the variables are exactly the same as before, but a new variable is required to get an instance of the `MatlabExternalApp` class. This variable is named `externalApp` and its type is then `jimc.MatlabExternalApp`. Figure 4.12 shows the variables defined for this simulation.

During the initialization of the simulation the connection to MATLAB has to be customized. Figure 4.13a shows this action. First, the variable `externalApp` has to get an instance of the `MatlabExternalApp` class. This is done by using the constructor of the class. Then the EJS variables `time`, `value`, and `frequency` have to be linked to the MATLAB variables `t`, `y`, and `f`, respectively. The linking is done by using the methods `setClient` and `linkVariables`. Note that the client is defined with the Java keyword `this` to set the owner of the Java variables. In the case of EJS the owner is always the object `this`. After linking, the `setCommand` method is used to indicate that the value of

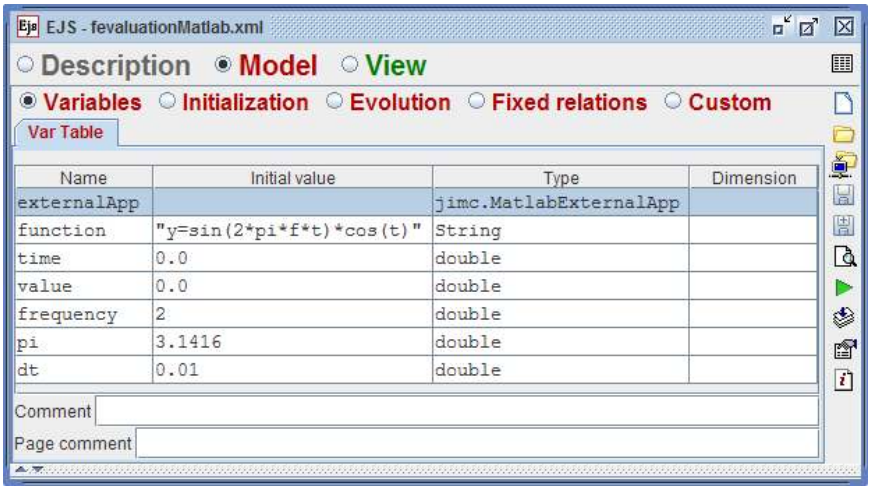


Figure 4.12: Declaring the variables in EJS.

the variable `function` will be evaluated as many times as the method `step` is executed. Finally, the connection with MATLAB is started by calling the method `connect`.

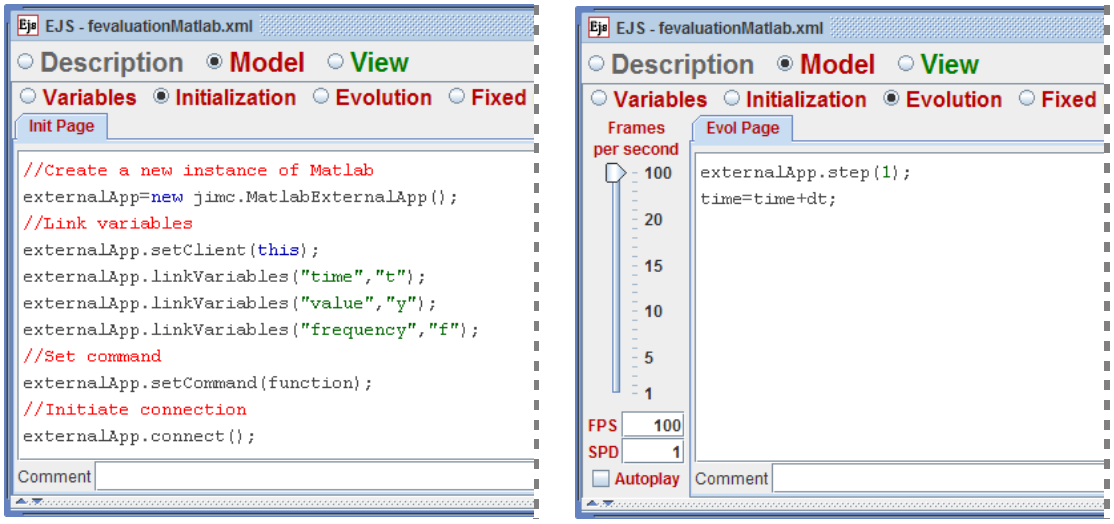


Figure 4.13: An example of using of `MatlabExternalApp` class of JIMC from EJS. a)Initializing the simulation b)Evaluating the function.

To evaluate repeatedly the command defined by `setCommand`, the `step` method has to be called from a page in the Evolution subpanel of EJS. In order to update the time, the same page is used to increase the time as in the first version of the example. The result of these actions is shown in Figure 4.13b. Note also that in this new version, the evaluation of the function is done only by MATLAB.

The design of the view of the original simulation is kept unmodified for this version with MATLAB. However, a new interaction must be added to the visual element `textField` in order to set a new MATLAB command, if the variable `function` is modi-

fied by the end user. This interaction is treated by the `Action` property, which executes the necessary Java code when end users modify the text of the variable `function`. Figure 4.14 shows the required modification of `Action` property.

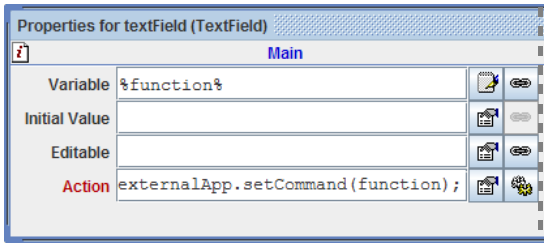


Figure 4.14: An example of using of `MatlabExternalApp` class of JIMC from EJS. The properties of the `textField` needed to capture the modification of the variable `function`.

Another modification to the original view needs to be added to finish the connection with MATLAB when end users close the simulation window. The `On Closing` property of the visual element `Frame` can be used for this purpose. This property executes the specified Java code when end users close the window, and in this case, the code to be used is simply the `disconnect` method.

4.2.2 Using the `RMatlabExternalApp` Java class from EJS

The class `RMatlabExternalApp` can also be used from EJS. In fact, the previous example can be used with minor modifications to perform a simulation with a JIM server.

The first modification is to declare the type of the `externalApp` variable as the class `jimc.RMatlabExternalApp`. This is done as before in the Variables subpanel of EJS.

The second modification is to get an instance of `RMatlabExternalApp` indicating a synchronous link, with the IP and Port number of the JIM server. This modification is accomplished with the following instruction:

```
externalApp = new jimc.RMatlabExternalApp("<matlab:IP:Port>");
```

After these modifications, end users can execute the simulation with the remote server without any other change. This is the advantage of using the synchronous mode of the remote link.

However, if end users report a slow simulation due to the network delays, authors could improve the performance of the simulation by using the asynchronous mode of the remote link. In this case, more modifications than for the synchronous mode are required.

First, as in the synchronous case, authors have to declare the type of the variable `externalApp` as `jimc.RMatlabExternalApp`.

Then, an instance of `RMatlabExternalApp` is obtained, but indicating now an asynchronous link. This modification is done by the following instruction:

```
externalApp = new jimc.RMatlabExternalApp("<matlabas:IP:Port>");
```

After that, the variable `function` needs to be modified in order to increase the time on the server side, i.e., the value of the `function` has to be now:

```
"y=sin(2*pi*f*t)*cos(t),t=t+0.01"
```

Accordingly, the code of the Evolution subpanel that increases the time has to be eliminated.

Finally, the `synchronize` method has to be used to respond to user interaction. This is required in the case of the slider that controls the variable `frequency`. To capture user interaction, the `On Release` property of this slider has to include the following Java code:

```
externalApp.synchronize();
```

With these modifications, the simulation works similarly to the synchronous case, but improving the performance over the network. Note that no further modification is required when the user modifies the value of the `function` variable (by interacting with the `textField` element). This is because the `setCommand` method defined in the property `Action` calls internally the `synchronize` method.

4.2.3 Using the `SimulinkExternalApp` Java class from EJS

The second example with EJS, shown in 4.1.2, can be considered again in order to describe how to use the `SimulinkExternalApp` Java class from EJS.

In this case, the original model, which is described by the ODEs system in (4.1), is replaced by the Simulink model of the bouncing ball presented in Figure 3.13a of Chapter 3.

As before with the `MatlabExternalApp` and `RMatlabExternalApp` classes, the variable `externalApp` has to be declared with the suitable type, in this case the type corresponds to the `jimc.SimulinkExternalApp` Java class.

Then, the connection with the external application is set again in the Initialization subpanel of EJS. The code required is shown in Listing 4.1. Similarly to the use of the `MatlabExternalApp` class, the connection with Simulink is set using the constructor of the `SimulinkExternalApp` class and the `setClient` and `linkVariables` methods. The connection is started by a call to the `connect` method.

```
1  //Create a Simulink connection
2  externalApp = new jimc.SimulinkExternalApp("bounce.mdl");
3
4  //Set location of the Java variables
5  externalApp.setClient(this);
6
7  //Link Java and Simulink variables
8  externalApp.linkVariables("position","bounce/Position","out","1");
9  externalApp.linkVariables("velocity","bounce/Velocity","out","1");
10 externalApp.linkVariables("elasticity","bounce/Elasticity","param","gain");
11 externalApp.linkVariables("gravity","bounce/Velocity","in","1");
12 externalApp.linkVariables("time","bounce","param","time");
13
14 //Start the connection
15 externalApp.connect();
```

Listing 4.1: Simulating the bouncing ball in EJS with JIMC, code to initiate the Simulink connection.

Obviously, this version of the simulation does not require the ODE editor of the Evolution subpanel. In this case, it is only necessary to call the `step` method to advance the time of the Simulink model. Thus, the code required in the Evolution subpanel in this simulation is simply:

```
externalApp.step(1);
```

Finally, the simulation has to capture the user interaction with the ball. This is necessary because when the user changes the position of the ball, the two integrators of the Simulink model need to be reset. The new values of the two integrators will be the velocity equal to zero and the position as given by the user. To reset the Simulink model, the `On Release` property of the `ball` visual element has to call the `synchronize` method. This modification can be seen in Figure 4.15.

The rest of the original simulation does not require any modification.

4.2.4 Using the `RSimulinkExternalApp` Java class from EJS

Similarly to the class `RMatlabExternalApp`, the Java class `RSimulinkExternalApp` has two modes of operation: synchronous and asynchronous.

Only two steps are required to turn the previous example into a simulation that uses the synchronous link. The first one is to define the type of the `externalApp` variable as

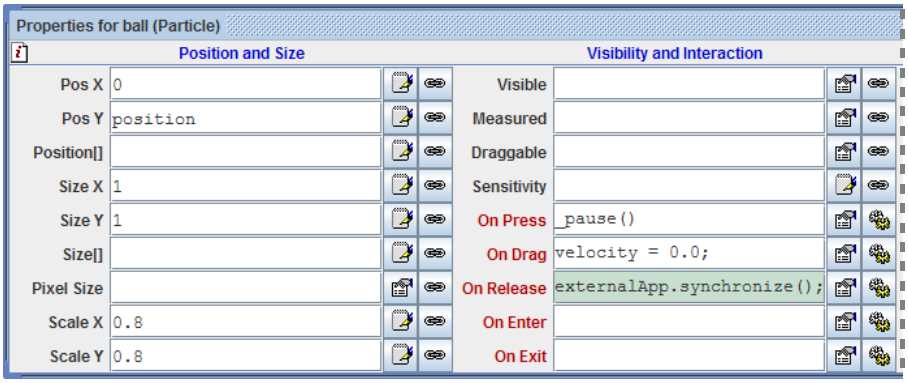


Figure 4.15: Resetting the Simulink model when the user releases the ball by calling the synchronize method.

`jimc.RSimulinkExternalApp`. The second modification is simply to get an instance of the `RSimulinkExternalApp` class, instead of `SimulinkExternalApp`, indicating the IP address and the Port number of the JIM server as follows:

```
externalApp =
  new RSimulinkExternalApp("<matlab:IP:Port>bounce.mdl");
```

These two actions allow authors to readily convert a local simulation into a remote version.

The asynchronous link can be selected however if end users report slow performance of the remote simulation. In this case, additional changes are needed. The first one is to define the type of the `externalApp` variable as `jimc.RSimulinkExternalApp`. The second one is to get an instance of the `RSimulinkExternalApp` class, indicating the IP address and the Port number of the JIM server, but selecting the asynchronous mode as follows:

```
externalApp =
  new RSimulinkExternalApp("<matlabas:IP:Port>bounce.mdl");
```

In order to capture the user interaction the `synchronize` method again needs to be used. This method was added to the local version of the simulation to execute a synchronization when the user releases the ball. Thus, the synchronization also needs to be added to the `slider1` and `slider2` sliders that control the gravity and the elasticity parameters. Similarly to the previous case, the following code needs to be added to the `On Release` property of both sliders.

```
externalApp.synchronize();
```

After these changes, end users should obtain a better performance of the simulation.

4.3 A built-in implementation of the ExternalApp in EJS

As described above, Easy Java Simulations can be used to create interactive simulations with standard engineering software. Authors can simply take Java packages, such as JIMC or others created by a programmer, that implement the interface `ExternalApp` for different tools.

However, observing the previous examples with JIMC, it can be concluded that a big part of the Java code required to connect the simulation with an external software, could be generated automatically with a suitable implementation of the `ExternalApp` Java interface directly in EJS.

This built-in implementation of the interface `ExternalApp` has been incorporated in EJS in order to ease, even more, the creation of highly interactive simulations that use engineering applications. For the time being, four engineering software are supported by this sophisticated feature of EJS: MATLAB, Simulink, Scilab, and Sysquake. Also, the remote connections of MATLAB and Simulink have been incorporated.

In order to connect EJS with an external application, authors have to create in the Variables subpanel a so-called `external page`. This special page is a new option of the Variables subpanel and is similar to a standard page to define variables to be used by the simulation. However, in this case, the page accepts the connection with a predefined external application, and also allows authors to link EJS variables to external variables.

The selection of the external application is done in the `External File` field. Here, the author sets one of the four external applications supported at this moment. The connections are defined for each external application with the corresponding strings. These strings are shown in Table 4.1.

In addition to the selection of the external application, authors can link variables to use the high-level protocol for the connection. This linking process is done simply by indicating the name of the external variable in the new `Connected to` column.

The methods supported by the implementations of the interface `ExternalApp` in EJS are now accessed by using the new predefined `_external` object. The way to use this functionality is much simpler than before. Firstly, is not necessary now to import any package, EJS imports all that is needed automatically. Secondly, the methods are

Table 4.1: The strings to set a connection with an external application in EJS.

Application	String	Description
MATLAB	<matlab>	Sets a connection with MATLAB.
Scilab	<scilab>	Sets a connection with Scilab.
Sysquake	<matlab>	Sets a connection with Sysquake.
Simulink	<matlab>mdlfile.mdl	Sets a connection with Simulink. The string *mdlfile.mdl* is the name of the Simulink model.
Simulink	<matlab(fixedtime)>mdlfile.mdl	Sets a connection with Simulink with fixed-time updated. The string `fixedtime` is a number used to speed up the simulation. The string *mdlfile.mdl* is the name of the Simulink model.
Remote MATLAB	<matlab:IP:Port>	Sets a remote connection with MATLAB. The IP and Port are the IP address and the Port number of the JIM server. The mode of the operation is **synchronous**.
Remote MATLAB	<matlabas:IP:Port>	Sets a remote connection with MATLAB. The IP and Port are the IP address and the Port number of the JIM server. The mode of the operation is **asynchronous**.
Remote Simulink	<matlab:IP:Port>mdlfile.mdl	Sets a remote connection with MATLAB. The IP and Port are the IP address and the Port number of the JIM server. The mdlfile.mdl is the name of the Simulink model. The mode of the operation is **synchronous**.
Remote Simulink	<matlabas:IP:Port>mdlfile.mdl	Sets a remote connection with MATLAB. The IP and Port are the IP address and the Port number of the JIM server. The mdlfile.mdl is the name of the Simulink model. The mode of the operation is **asynchronous**.

called in a similar way as before but now using the `_external` object. For instance, to start the connection with an external application the method `connect` has to be used as follows:

```
_external.connect();
```

The next sections describe in more detail how authors can connect EJS with a predefined external application.

4.3.1 A built-in feature to connect MATLAB with EJS

In order to show how to use the new feature provided by EJS to connect with MATLAB, the example described in Section 4.2.1, evaluating a function in MATLAB, is revisited.

As explained above, first of all a new external page of the Variables subpanel needs to

be created. In this page, the string < *matlab* > that sets the connection with MATLAB is entered in the `External File` field.

The next step consists in linking client and external variables. Thus, the client variables `time`, `value`, and `frequency` need to be linked to the external variables `t`, `y`, and `f`. But, instead of using the `linkVariables` method, these connections are set in the `Connect to` column for each client variable.

After all modifications are done, the Variables subpanel of this new version should look like in Figure 4.16.

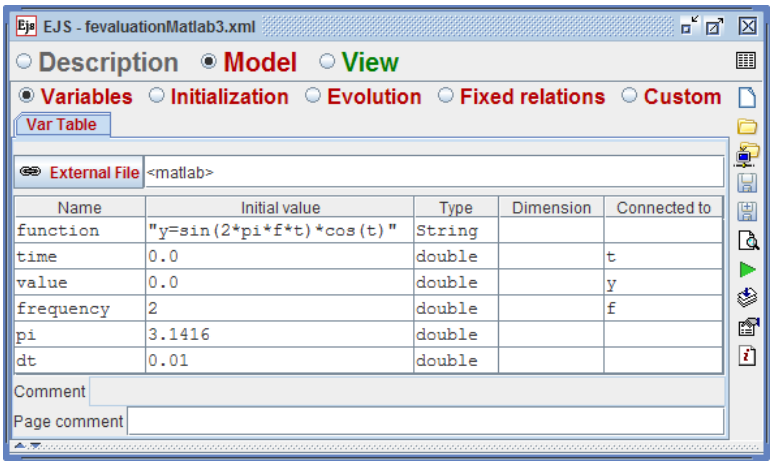
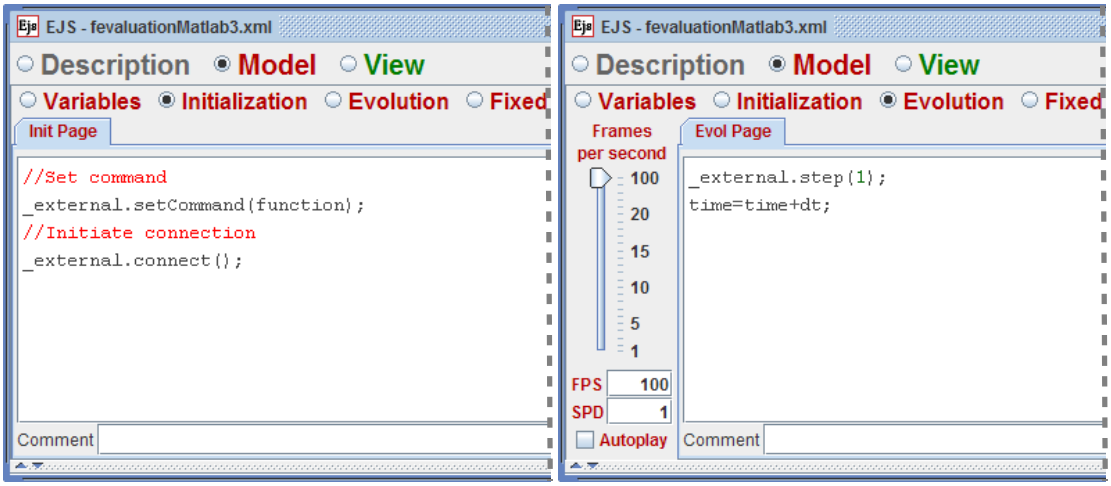


Figure 4.16: Setting external connection with EJS.

This way of setting the external application and linking the variables avoids writing the Java code required to get an instance of the `MatlabExternalApp` class and to link the variables, shown in Figure 4.13a. In this new version, the page of the Initialization subpanel is just used to set the function to be evaluated and to start the connection with MATLAB. This is done using the `_external` object instead of an instance of the `MatlabExternalApp` class as shown in Figure 4.17a.

Once the connection with MATLAB is established, the simulation updates the values of the linked variables and executes the command defined, by calling the `step` method from the `_external` object as shown in Figure 4.17b. Note that the code used here is similar to that shown in Figure 4.13b but, instead of using an instance of the `MatlabExternalApp` class, the `_external` object is used. Obviously, this new way to call the methods of the interface `ExternalApp` needs to be used wherever a simulation uses this built-in feature of EJS.



a) b)

Figure 4.17: An example of using of the built-in implementation of `MatlabExternalApp` class in EJS. a) Initializing the simulation b) Evaluating the function.

4.3.2 A built-in feature to connect a remote MATLAB with EJS

The remote connection to MATLAB is also supported by EJS. The strings defined in Table 4.1 show how to use this feature. Both modes of operation can be set simply by using the suitable string in the `External File` field. The linking among client and external variables is similar to the process described for the local case. Some examples of this remote connection can be found in (Farias, Dormido, Esquembre, Vargas & Dormido-Canto 2008, Farias, Keyser, Dormido & Esquembre 2009, Farias et al. 2010).

All the commands used in the local connection work similarly to the remote case described in Section 4.2.2 to use JIMC with EJS.

4.3.3 A built-in feature to connect Sysquake and Scilab with EJS

Easy Java Simulations also provides a built-in implementation of the Java interface `ExternalApp` for the software Sysquake. The way to use Sysquake is the same as described before for MATLAB. In fact, the previous simulation can be easily transformed for use with Sysquake by just replacing the string <`matlab`> of the `External File` field (see Figure 4.16) with the corresponding string for Sysquake (<`sysquake`>). No other change is required to run the simulation with Sysquake.

The same modification can be done to transform the simulation to be used with Scilab. However, it is also necessary to modify the string "$y = sin(2 * pi * f * t) * cos(t)$" variable `function` to the string "$y = sin(2 * \%pi * f * t) * cos(t)$". This is needed because

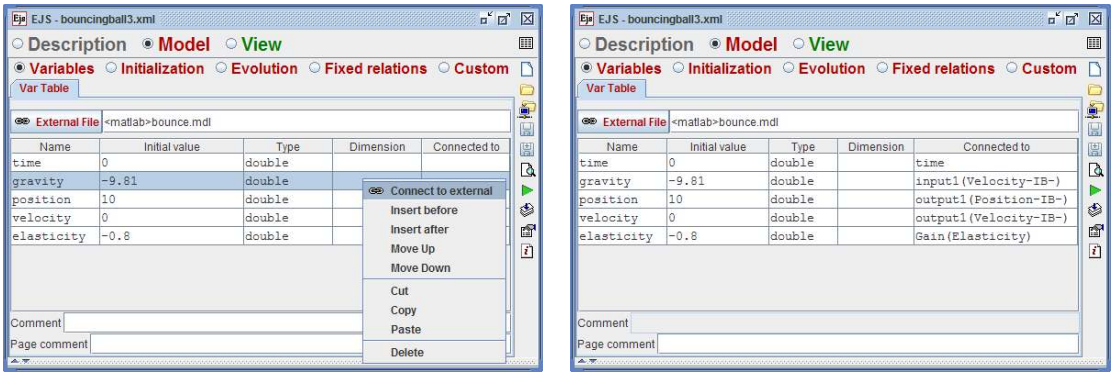
the constants in Scilab must be written with the % prefix. Apart from this change, no other modification is required to the simulation in order to execute it with Scilab.

4.3.4 A built-in feature to connect Simulink with EJS

In order to describe the connection between EJS and Simulink using the built-in feature, the example commented in Section 4.2.3 is analyzed again.

Similarly to what has been done, the connection with Simulink is set in the `External File` field of an external page in the Variables subpanel of EJS. The string to indicate this connection was <`matlab`>`mdlfile.mdl` as described in Table 4.1. In this case, the `mdlfile.mdl` parameter is replaced with the name of the model to be used, `bounce.mdl`. The string `mdlfile.mdl` also accepts a relative path of the Simulink model if it is not located in the same directory where the simulation is. This path can also be obtained by clicking on the `External File` button, which opens a file selection dialog to select the Simulink model from any location in the file system.

The linking between client and external variables is done again by using the `Connect to` column. However, for Simulink connections, authors can indicate in a much more sophisticated way the link between an EJS variable and an input/output or parameter of a block of the Simulink model. The process starts by right clicking on the row of the variable to be connected. This opens a pop-up menu where the option `Connect to external` has to be selected. Figure 4.18a shows this pop-up menu for the connection of the `gravity` variable.



a) b)

Figure 4.18: An example of use of the EJS built-in implementation for Simulink. a) Setting the external application b) The result of the variables linking.

After selecting the option `Connect to external` from the menu, a connection dialog appears to select the external variable (i.e., an input/output or parameter of a block)

from the Simulink model. This dialog is shown in Figure 4.19a.

The connection dialog displays initially the global variables and parameters of the model, such as `time`. The connection to an external variable of any block of the model can be done by either selecting the block in the combo box included in the dialog or, more naturally, displaying the Simulink model itself (clicking on the check box on the upper-left corner of the dialog) and directly clicking on the desired block. The connection dialog will then display the variables and parameters of the block selected.

The connection dialog contains three white areas in its upper half part. These areas list separately input variables, output variables, and parameters of the selected block. To select one of the variables, it is only necessary to double click on it and it will be added to the corresponding area in the lower-half-part of the dialog. Unselecting is done in a similar way.

The category from which the variable is selected has an effect on what can be done with it:

- Input variables can be freely changed from EJS. That is, any of the mechanisms of EJS can be used to change them, at any time, either in the model or in the view. Any change in these variables will be immediately reflected in the Simulink model.

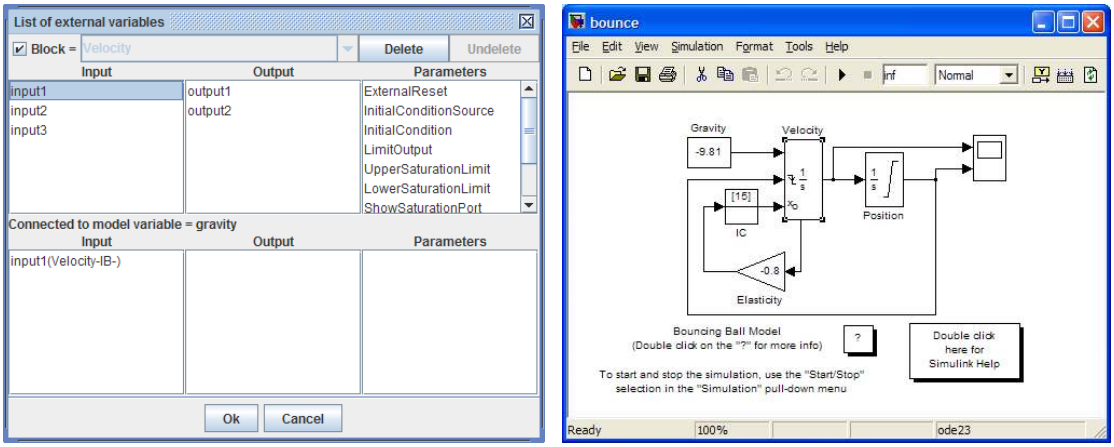
- Output variables can, on the contrary, only be read by EJS. Hence, any change executed to their associated variables in EJS will not affect their value in the Simulink model. Note that there is an exception with the integrator blocks, which can be changed using the reset mechanism.

- Finally, parameters can also be changed from EJS.

For instance, the `gravity` variable needs to be connected with the first input of the `Velocity` block. This is done by selecting this block on the Simulink model and then double clicking the external variable `input1`. Figure 4.19 shows this process. Finally, clicking the button OK establishes the link.

The final result of the linking between client and external variables is shown in Figure 4.18b.

Once the connection and linking of the variables is done, the next step is to start



a) b)

Figure 4.19: Connection dialog to link client and external variables to the Simulink model. a)The dialog b)The Simulink model.

the connection with Simulink in a page of the Initialization subpanel. Thus, the code in the Initialization page is simply:

```
_external.connect();
```

Compare this code with the code required by the example of Section 4.2.3 shown in Listing 4.1. Observe that, now, only the `connect` method is needed in the Initialization page, since the connection and linking of the client and external variables was done previously in the Variables subpanel of EJS.

The rest of the simulation is similar to the example of Section 4.2.3. Again, the methods have to be called by using the `_external` object instead of using the instance of the `SimulinkExternalApp` class (i.e., the `externalApp` variable).

The `deleteBlock` method and the `<matlab(fixedtime)>mdlfile.mdl` string can also be used in order to speed up the simulation. This last option has to be used by replacing the `fixedtime` string with the value of the *sample time* desired.

A more detailed description of the use of Simulink models from EJS can be found in (Dormido, Esquembre, Farias & Sánchez 2005).

4.3.5 A built-in feature to connect a remote Simulink with EJS

The synchronous and asynchronous modes for a remote connection with Simulink are also supported by EJS. The use of either of these modes depends only on the string written in the `External File` field. However, there is a slight difference from the connection with a local Simulink when the variables are linked. The connection with an

external variable of any block of the model can be done only by selecting the block in the combo box included in the dialog, since the Simulink model is open in the remote server and no blocks can be clicked by the author. However, if the check box of the dialog is ticked then an image of the Simulink model stills appears in order to facilitate the recognition of the role of the blocks in the model. Although this option can help authors in the linking process, the recommendation is simply to set the connection with a local Simulink and then change the connection to a remote server.

Once the connection and linking is finished, a similar way to initiate the connection and control the execution of local Simulink models can be done for a remote simulation. Once more, the `synchronize` method needs to be used to reset the integrators and to synchronize the asynchronous simulations. Some examples of the use of this remote connection can be found in (Farias, Esquembre, Sánchez, Dormido, Vargas, Dormido-Canto, Dormido & Duro 2006*b*,*a*, Fabregas et al. 2010).

4.3.6 Other built-in features in EJS

Easy Java Simulations has also implemented two nice, very useful features to perform special simulations.

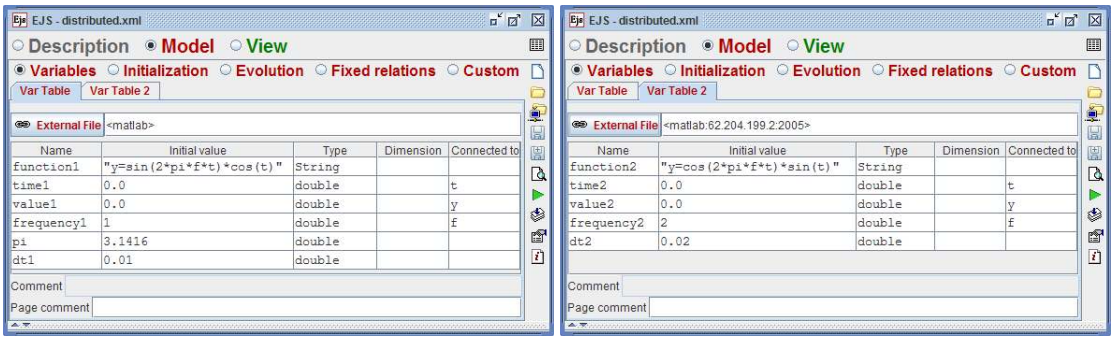
The first feature is the implementation of the `setAlternative` method. Using this method authors can provide a second engine to run a simulation which uses an external application. Thus, for instance, a simulation designed to be executed with a local MATLAB can be optionally executed using a remote MATLAB if the computer of the end user lacks a MATLAB installation, but a remote server is available. The local MATLAB is still preferred to a remote MATLAB for performance reasons but the remote will be used as a safe resource. To support this option, the author should use the following code:

```
_external.setAlternative("<matlab:IP:Port>");
```

This code has to be executed before the `connect` method is called. Additionally, alternative applications (e.g. other remote servers) can be added by repeated use of the `setAlternative` method.

The second feature implemented in EJS allows authors to control more than one external application. This is easily done by simply creating more than one external pages in the Variables subpanel of EJS. Thus, each external page can be associated with

a different external application. For instance, it is possible to modify the example of Section 4.3.1 to compute two functions, one function in a local MATLAB and another function in a remote MATLAB.



a) b)

Figure 4.20: A simulation setting the connection with two external applications. a) The connection with a local MATLAB. b) The connection with a remote MATLAB.

Figure 4.20 shows the two external pages created. The `Var Table` page sets a connection with the local MATLAB, and the `Var Table 2` page sets a connection with the remote MATLAB. Observe that two groups of similar variables in EJS are connected to the respective external variables. Thus, for instance the variables `function1` and `function2` are used to set the command to be evaluated in the local and remote MATLAB, respectively.

The manipulation of the external applications can be done in two different ways: globally or separately. The first way allows to control all the applications at the same time, and the second way allows to controls a specific external application.

The global manipulation is done simply by using the `_external` object as described above. To specifically manipulate one of the applications, its name has to be used. The name of the application is the string used to set the link in the external page of the Variables subpanel of EJS.

For instance, to start the connection with both external applications, the following code has to be used:

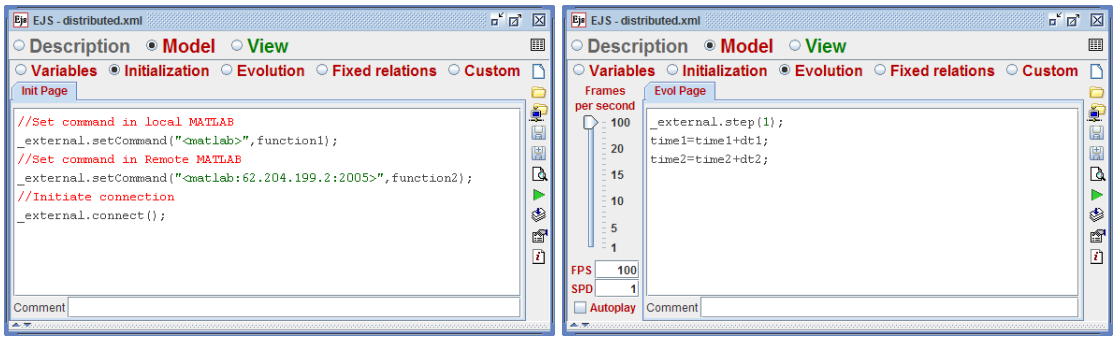
```
_external.connect();
```

To only start the connection with the local application, the following code can be used:

```
_external.connect("<matlab>");
```

In all the methods supported by `_external`, if the specific manipulation is required, the first parameter has to be used to indicate the name of the application.

Figure 5.22a shows the initialization page of the simulation. In this case, both applications are connected at the same time. However, each command to be evaluated is set to its respective `function` variable.



a) b)

Figure 4.21: Manipulating two external applications. a) Initialization page b) Evolution page.

The Figure 4.21b shows the evolution page of the simulation. Note that here, both applications are stepped at once.

The possibility of using more than one applications is useful for example to distribute complex calculations in two or more machines. This gives authors the possibility to use parallel algorithms in their simulations.

4.4 Building remote laboratories with MATLAB and EJS

Traditional laboratories (labs for short) are a fundamental pedagogical tool for the education of engineering students. In particular, control engineers need to be in touch with real equipments (also called plants) to apply all the theory learnt in university lectures. However, traditional labs present temporal and geographic constraints that normally make it impossible to access these pedagogical resources at any time and even less from any location.

Remote laboratories have been increasingly used in control engineering education (Gillet et al. 2000, Gomes & Bogosyan 2009, Leva & Donida 2008). The main reason being the opportunities that technologies of information and communications provide nowadays. Great capabilities of graphical computing, interaction and networking make it possible to bring to students new ways to experiment with real plants without time and location constraints.

However, taking advantage of these fascinating possibilities is not easy for most instructors, who are normally non programming experts. This is why specialized authoring tools are highly needed. In this section, a new approach to ease the building of remote labs is described. The approach is different to other alternatives (Leva 2006, Duro et al. 2008, Lazar & Carari 2008, Vargas et al. 2008) because instructors can use the de-facto standard software MATLAB as the main tool to control the real plant.

The two main software tools used are MATLAB and EJS, which have both been described above. On the one hand MATLAB will be used to implement all the technical aspects to control the real plant. On the other hand, the Easy Java Simulations authoring tool is used to simplify the creation of an interactive human interface for the remote lab. Further details about this approach can be found in (Farias, Keyser, Dormido & Esquembre 2009, Farias et al. 2010).

Commonly, remote labs come with predefined controllers that can only be tuned or mixed by end users in order to produce a customized controller, see for example (Leva & Donida 2008, Lazar & Carari 2008, Gomes et al. 2007, Duro et al. 2008, Gillet et al. 2005, Vargas et al. 2006). However, and taking advantage that MATLAB script is an interpreted language, the approach presented here enhances the customization, allowing also students to define on-the-fly their own control algorithm, which provides great flexibility for the designing of many control strategies.

4.4.1 Types of remote labs

A remote control labs actually consists in the remote operation of equipment. This implies considering a network one component of the control of a real plant.

Two main groups of remote labs can be considered depending on whether the network is part of the control loop or not. If the network is considered, then the remote lab is called **Networked Control Lab**, otherwise the remote lab can be denominated **Remote Monitoring Control Lab**.

In Figure 4.22, a scheme of a Networked Control Lab is shown; here, the controller is located at the client side, which means that the network effects are taken into account for control purposes. If the controller is located at the server side, then network effects are not considered, the control of the real plant can be much easier, and the client side application is used only for monitoring the control of the real plant. Common and

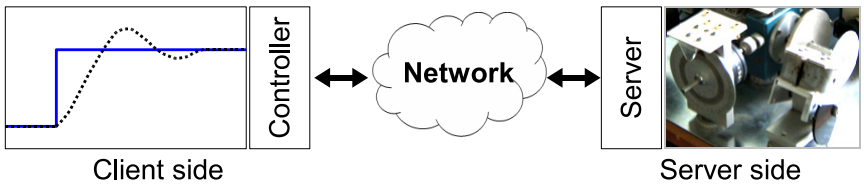


Figure 4.22: A networked control lab. On the client side an application is used to monitor the state of the plant and to compute the control action. On the server side, the real plant is manipulated according to the controller outputs.

particular aspects for both types of remote labs are described in more detail in the next subsection.

4.4.2 Common aspects of implementing remote labs

The approach presented here can be summarized in Figure 4.23. As this figure shows, other elements are required in order to make the remote lab available on-line for students. Among others, at least a web-server, an IP camera, and a Data Acquisition Card (DAQ Card) are needed.

The web-server is required if authors want to make the remote lab available as an applet. A web-server will also be needed if the teacher adds some simulation applets in the web-site. A good option to provide web access to students is to use the open source software called Apache Tomcat. This web-server also offers security features to allow use only to authorized students of a remote lab. A second level of security can be added in order to provide a basic booking system of the lab. This feature could be implemented in the application itself using the authorization feature of JIM server and the `connect(user,pwd)` method.

To show students a view of the real plant, a simple web camera can be used. However, it is preferable to use an IP camera because of its built-in web server that can stream video images directly to the Internet. Moreover, EJS has a specific visual element to display video from stream servers, so the access to the streams of IP cameras is quite direct and simple to use in EJS.

There are many options for developers who require controlling external hardware (plants) from computers. In this case, different data acquisition(DAQ) cards can be used, the only restriction in this approach is that the selected card has to be compatible with MATLAB. The communication process with the DAQ card can be done using the Data Acquisition toolbox. This toolbox is a collection of M-file functions and DLLs that

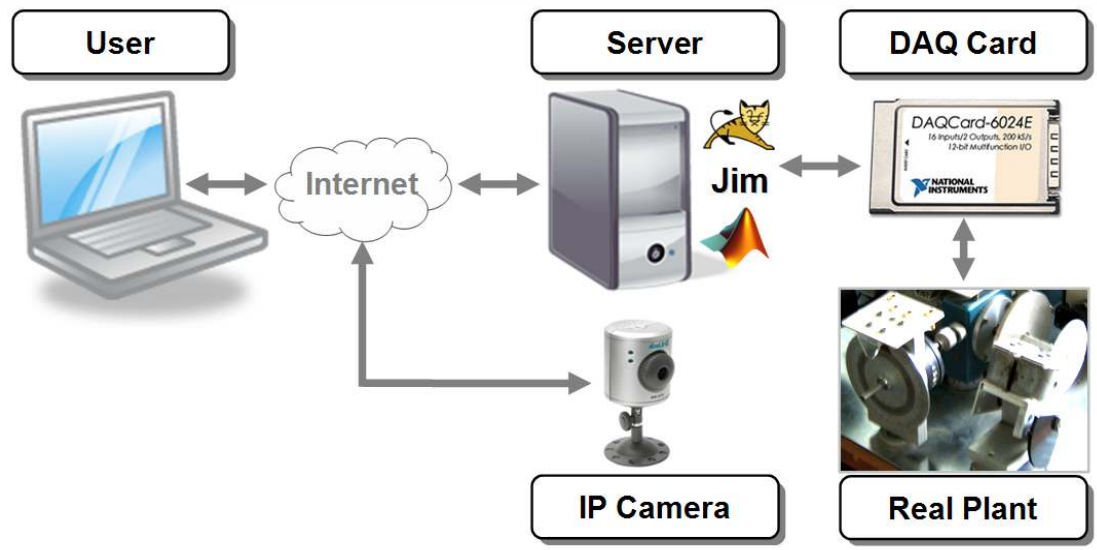


Figure 4.23: Elements required for a networked control lab implemented with the MATLAB-EJS approach.

enable users to interface with specific hardware. The toolbox provides users with these main features:

- A framework for bringing live, measured data into the MATLAB workspace using PC compatible, plug-in data acquisition hardware.
- Support for analog input (AI), analog output (AO), and digital I/O (DIO) subsystems including simultaneous analog I/O conversions.
- Support for popular hardware vendors/devices.

For further information about the toolbox see (The MathWorks 2007).

Acquisition Process from DAQ Cards

A typical code to initiate the data acquisition process using the toolbox of MATLAB is shown in code Listing 4.2. In this case, there are two input channels and also two output channels. The input channels are created to read the position and the speed of a servo motor. One output channel is used to send the command signal to the motor, and the other one is used to feed the position sensor (a potentiometer). After the channels have been created, it is necessary to configure the DAQ card to continuously acquire the data.

After the initialization is completed, the channels can be read using:

```
%Create input channels
AI=analoginput('nidaq','1');
Speed=addchannel(AI,1);
Position=addchannel(AI,0);
set(AI,'InputType','SingleEnded');
AI.Channel.InputRange=[-10 10];

%Create output channels
AO=analogoutput('nidaq','1');
VoltOutSpeed=addchannel(AO,0);
VoltOutPosition=addchannel(AO,1);

%Configurate sampling and trigger
set([AI,AO],'SampleRate',1/0.05);
set(AI,'TriggerType','Immediate');
set(AO,'TriggerType','Immediate');

%Acquire data continuously
set(AI,'SamplesPerTrigger',inf);

%Send out initial volts
putdata(AO,[0,1]);
start([AO AI]);
```

Listing 4.2: Initiate data acquisition process.

`peekdata (AI, numberOfInputChannel);`

and written using:

`putdata(AO, numberOfInputChannel);`

Notice that the code to read and write channels can be executed from EJS using the methods `eval` and `getDouble`. Listing 4.3 shows the Java code used to read and write the input/output channels. Notice that the code required to interface the DAQ card is sent to MATLAB from the application using the `_external.eval(String)` built-in function.

```
public double[] getData(){
 double[] data;
 //Read input channels
 _external.eval("s=[toc,peekdata(AI,1)];");
 data=_external.getDoubleArray("s");
 return(_data);
}

public double sendOut(double cmd){
 //Saturation of command signal
 cmd=commandLimits(cmd);
 //Write output channels
 _external.eval("putdata(AO,["+cmd+",1])");
 _external.eval("start(AO)");
 return(cmd);
}
```

Listing 4.3: Read and Write the channels from EJS.

On the selection of a type of remote laboratory

A few additional considerations are required when the particular type of remote lab is implemented.

First of all, the real plant has to be selected carefully, since even for remote monitoring control labs, the time constant of the plant cannot be too small. A fast plant can generate a lot of data that can be difficult to visualize and transport over the network. This could produce a poor user experience in the client application used by the students. Therefore, slow plants are preferred to fast ones for remote labs.

Depending on where the controller is located, the main loop for controlling the plant will be different. In networked control labs, the main control loop of the application is done at the client side, since in these labs the control signal is computed there. This requires adding the code of the main loop on the model panel of the EJS application.

The main loop of the networked control lab is displayed in Listing 4.4. The code presented is part of the EJS application and has to be located in an evolution page of the model panel in order to execute the code continuously. The code has three main parts: open loop, closed loop, and getting data. In the open loop mode, the input, given by the user (`commandUser`), is sent directly out to the real plant. In the closed loop mode, the control signal is computed using the `controller` function, and the signal is then sent out to the motor. Finally, once the open or closed loop stages have been executed, the data is read from the sensors of the plant. Notice that the functions for reading and writing the channels (`getData` and `sendOut`) have been defined earlier in Listing 4.3.

```
1  //Open Loop
2  if (isOpenLoop) {
3   commandUser=sendOut(commandUser);
4   inputApplied=commandUser;
5  //Closed Loop
6  }else{
7   if (speedControl) cs=controller(reference,speed);
8   else cs=controller(reference,position);
9   controlSignal=sendOut(cs);
10  inputApplied=controlSignal;
11 }
12 //Getting data
13 data=getData();
14 position=data[2];
15 speed=data[1];
16 time=data[0];
```

Listing 4.4: Main loop of the remote lab in the EJS application.

On the contrary, in the remote monitoring control labs, the main control loop of the application is done on the server side. For this purpose, authors can use the feature of

Program Scheduling of MATLAB (by using timers)(The MathWorks 2009*b*) to compute continuously the control action on the server side. This way, the client side is used only to modify some control parameters or to show the main signals. An example of timer object to compute the control action by calling periodically (every 50ms) an M-function (computeCS) is the following:

```
t = timer('TimerFcn','computeCS',
          'ExecutionMode','fixedRate','period',0.05);
```

Finally, with respect to the synchronous and asynchronous links of JIM, the choice between both options depends also on the kind of remote lab that the instructor wants to implement. On the one hand, an asynchronous link is preferred if the controller is located on the server side, which means that the lab is used mainly to monitor the remote plant. In this case, the network effects are not interesting from the control point of view, and therefore, by using the asynchronous link, the network delays are reduced considerably in comparison to the synchronous link.

On the other hand, if the controller is located on the client side, the synchronous link has to be used, since the network effects must be considered.

4.4.3 Networked control lab implemented

The lab discussed here is a Java application designed to control a simple servo-motor through the Internet. This lab uses a real servo-motor of FEEDBACK located at Ghent university in Belgium. The selected DAQ card was the 6024E card manufactured by National Instruments. In Figure 4.24, a scheme of the implemented lab is presented. Notice that the continuous-time system is controlled by a discrete-time controller with a sampling period of 50 ms. The networked control lab is divided in three sections: the client side, the network, and the server side.

The implemented lab uses two MATLAB sessions, one located on the client side to compute the control signal, and another one located on the server side, to interface the servo motor through the DAQ card 6024E. At each sampling time, the JIM server collects data from the plant by using the MATLAB engine at the server side. Then, JIM server sends data to the EJS application by using standard TCP/IP Java methods. The application uses the data and the local MATLAB engine to compute the control signal by executing a given MATLAB script (which can be modified on line by the student).

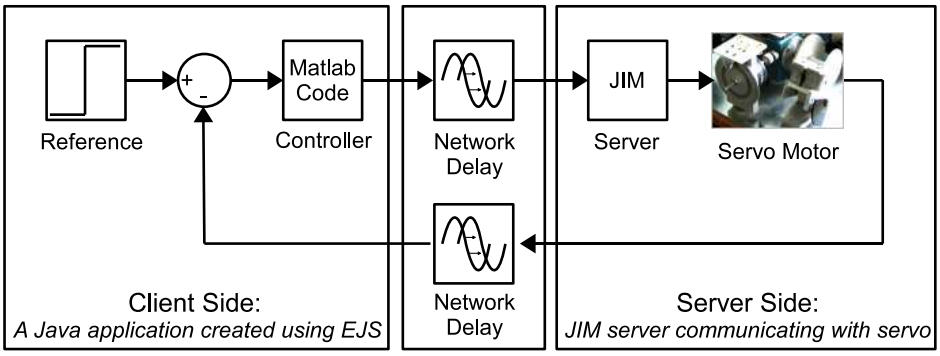


Figure 4.24: Scheme of the implemented remote lab.

Then, the application retrieves the value of the control signal from the local MATLAB engine and sends it via TCP/IP to the JIM server. Finally, the JIM server sends out the computed control action to the servo motor by means of the server MATLAB engine.

4.4.4 Computing the control signal

As previously mentioned, the lab uses a local connection to MATLAB to evaluate the control action. This means that students can write on line the MATLAB code to control the plant. The code used by default (a PID controller) is shown in Listing 4.5. Here, a MATLAB function named `computeControl` will be called from the `controller` function of Listing 4.4.

The `computeControl` function has five input and one output variables. The input variables are `r` (set point or reference), `y` (controlled variable for position or speed of motor), and three optional variables `a`, `b`, and `c` (in this case, these free variables are used as gain, integral time, and derivative time of the PID controller). The output variable `cs` is returned to the `controller` function, in order to send out the control action to the real plant (see Listing 4.4).

4.4.5 Model of the remote servo motor

From the control point of view, the model of the networked control lab can be divided in three main parts: a model for the network delay, and models for the speed and position of the servo motor, see Figure 4.25.

It can be assumed that all delays from the network can be represented by a single variable dead-time, which can have different values depending mainly on the distance between server and client and the traffic overhead in the network. The speed model is a first-order system (FOS) plus two non linearities: a saturation (limiter) and a dead-zone.

```
function cs=computeControl(r,y,a,b,c)
 %r=reference y=output [a b c]=EJS parameters
 %cs=control signal
 %Initialization
 persistent Iold Dold yold
 if isempty(Iold)
  Iold=0;Dold=0;yold=0;
 end
 %Calculate Control Signal
 beta=1;N=10;h=0.1;
 P = a*(beta*r-y);
 I = Iold;
 D = c/(N*h+c)*Dold+N*a*c/(N*h+c)*(yold-y);
 cs = P + I + D;
 Iold = Iold + a*h/b*(r-y);
 Dold = D;
 yold = y;
```

Listing 4.5: Default code to calculate a PID controller.

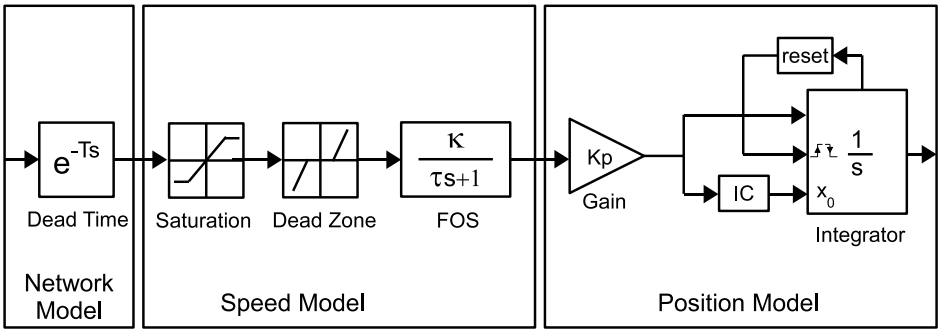


Figure 4.25: Model of the networked control lab. The remote servo motor is divided in three parts: the network model, the speed model and the position model.

The FOS system represents the dynamics of the amplifier, the motor, the brake and the tachometer of the servo motor plant. Regarding the dead-zone, the range detected in the real plant is located in $[-\delta_1, \delta_1]$, where $\delta_1 = 0.1V$. In the case of the saturation of the amplifier, it allows input values into the range $[-\delta_2, \delta_2]$, where $\delta_2 = 0.2V$.

To get the position of the servo motor, a potentiometer was used. The potentiometer is fed by the DAQ Card with $1V$. So, the voltage obtained indicates the position of the motor. This position model can be represented by a gain plus a pure integrator. However, in order to have a better model of the real plant behaviour, an automatic reset can be added to the pure integrator to limit and handle the discontinuity of the servo position sensor, which produces a signal between $0V$ and $1V$ to represent the angular position from $0°$ to $360°$.

The motor dynamics can be modified by means of the position of the magnetic brake. Normally two positions are used: *partial brake* and *full brake*. For instance, when *full brake* is selected, then the gain κ and the time constant τ are smaller than when the

partial brake is used. In the *full brake* case, typical values for κ and τ are about 10 and $1s$ respectively. On the other hand, in the *partial brake*, normally the values for κ and τ are about 14 and $2.5s$ respectively.

4.4.6 Control aspects of the remote servo motor

There are two control objectives regarding the implemented lab configuration: speed control and position control. However, in this work only the position control will be further analysed, since it is more difficult than speed control. In a feedback control of the remote servo, the model of the motor can be considered a second order system with a dead-time, a dead-zone, and a saturation. The effect of each part of the model will be discussed now:

Dead-Zone

Consider a second order system composed only by the FOS model plus a gain (k_p) and a pure integrator. If a P controller (k_c) is also considered, then the open loop transfer function of this simple model is given by Equation (4.2), where $K = k_c \cdot \kappa \cdot k_p$.

$$GH = \frac{K}{s(\tau s + 1)} \tag{4.2}$$

From the control point of view, this second order system can be controlled using only the P controller. However, if the dead-zone is taken into account (see Figure 4.26a), then the control requires an integral action to eliminate the steady-state error. Otherwise, the system response of the remote lab can be similar to the plot presented in Figure 4.27.

The steady-state error is demonstrated by analyzing the obtained error when the control signal ($cs = k_c \cdot e$) is within the dead zone ($-\delta_1 \leq cs \leq \delta_1$). In this situation, the output of the dead-zone block is zero, so the control loop is broken and therefore the position signal is just a constant. It can be demonstrated that in this case the steady-state error (e_{ss}) is different from zero and it is located in a band (see Equation (4.3)). The extremes of this band depend on the dead-zone limits and the gain of the proportional controller.

$$\frac{-\delta_1}{k_c} \leq e_{ss} \leq \frac{\delta_1}{k_c} \tag{4.3}$$

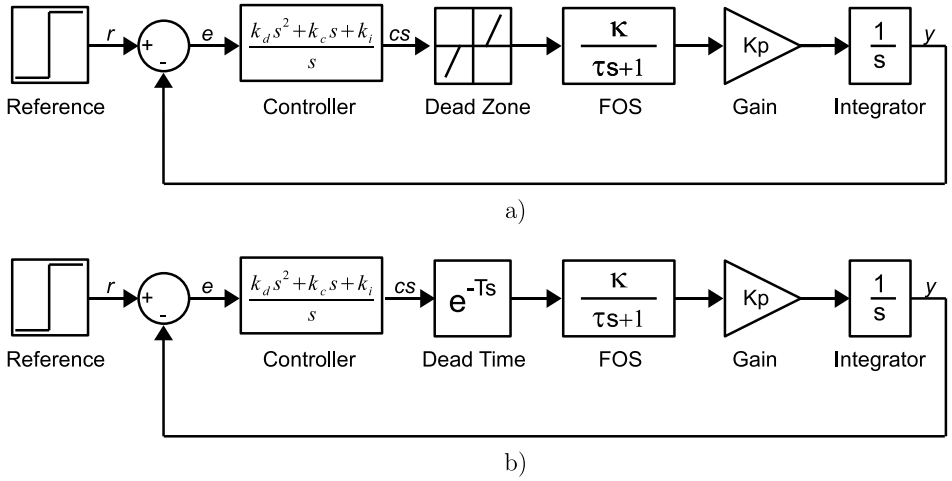


Figure 4.26: Remote servo as a second order system plus dead-zone a) and as a second order system plus dead-time b). Note that the block "Controller" represents a PID controller, which can be computed in the remote lab by the `computeControl` MATLAB function described in Listing 4.5.

Dead-Zone and Saturation

Consider the previous case plus the saturation. Here, if the saturation is also taken into account, then the integral action required to eliminate steady-state error, has to consider the limits of the saturation block. So, if a PI controller is selected, then an anti-windup scheme should be applied to avoid a slow control action (Åström & Hägglund 2005).

Dead-Time

If only the linear parts of the motor and the dead-time are considered (see Figure 4.26b), then a second order system plus dead-time is obtained. A detailed description of dead-time processes can be found in (Normey-Rico & Camacho 2007). In this situation, the network delays can strongly affect the control performance. Here, different solutions can be applied; one of them could be the use of a Smith predictor if the delay remains approximately constant during the remote control session. However, if the delay connection is highly variable, then more elaborated algorithms are required. To simplify the problem here, this work will focus on fixed time-delay systems. Further analysis of varying time-delay can be found in (Normey-Rico & Camacho 2007, Cristea et al. 2005, Liu et al. 2007).

If the network delay is approximately constant and a P controller is used, it is possible to calculate how much delay the system can tolerate before becoming unstable (Dorf & Bishop 2004). To calculate this margin delay firstly, the open loop transfer function of

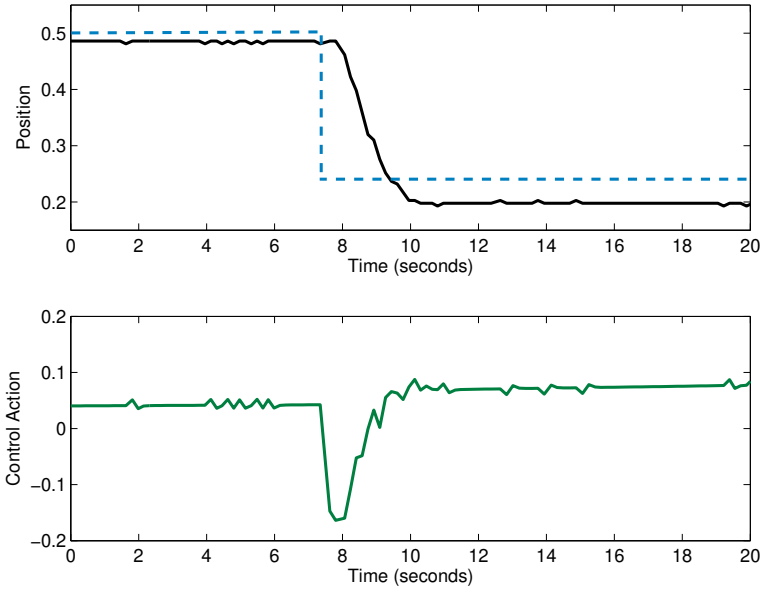


Figure 4.27: System response of the remote lab implemented using a proportional controller. The upper plot shows the output (position) and the set point of the lab. The lower plot shows the control action. In this experiment, the control performance is being affected mainly by the dead-zone. Note that saturation is not limiting the control action. The network delays are reduced by using the lab through a LAN network.

the model is computed, which in this case is given by Equation (4.2).

Secondly, computing the magnitude and the phase margin of the transfer function given by Equation (4.2), and assuming that $\frac{1}{\sqrt{\tau K}} \approx 0$ (which is a common situation when partial or full brake are selected), some useful approximations for ω_{gc} (gain cross over frequency) and ϕ_m (phase margin) are obtained. These approximations are given by Equations (4.4) and (4.5), respectively.

$$20 \log \frac{K}{\omega_{gc}\sqrt{\omega_{gc}^2\tau^2 + 1}} = 0 \rightarrow \omega_{gc} \approx \sqrt{\frac{K}{\tau}} \tag{4.4}$$

$$\phi_m = \pi - \frac{\pi}{2} - \arctan \omega_{gc}\tau \rightarrow \phi_m \approx \frac{1}{\sqrt{K\tau}} \tag{4.5}$$

Finally, using ω_{gc} and ϕ_m, the delay margin (T) can be calculated using (4.6). Therefore, this last equation indicates that the delay margin is inversely proportional to the product of $k_c \cdot \kappa \cdot k_p$.

$$\phi_m = \omega_{gc}T \rightarrow T \approx \frac{1}{k_c \kappa k_p} \tag{4.6}$$

Notice that even small delays could lead to an unstable closed loop. For example, when the full brake is selected ($\kappa = 10$ and $\tau = 1$) and assuming that $k_p = k_c = 1$,

the delay margin is approximately $0.1s$, which is a usual value for Internet delays. This result implies that, if the system has less brake then it is less stable. Obviously, this delay margin can be modified by changing the value of k_c in the P controller.

Dead-Time, Saturation and Dead-Zone

Consider the linear model of the motor plus the dead time, saturation and the dead zone controlled by a P controller. In this situation, the system can produce limit cycles if there is enough dead time as Figure 4.28 shows.

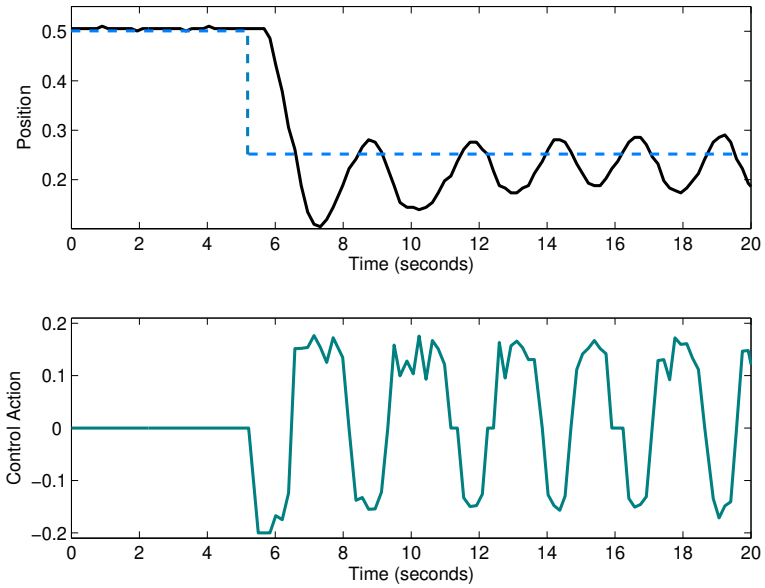


Figure 4.28: System response of the remote lab implemented under high network delays (> 0.2 seconds). The upper plot shows the output (position) and the set point of the lab. The lower plot shows the control action. In this experiment the control performance is being affected by all non-linearities. If the network delays are high enough then the system response is a stable limit cycle. Here, the remote lab located at Ghent University (Belgium) was tested from UNED (Spain).

To carry out an approximate limit cycle study of this nonlinear system, the non-linear elements can be characterized by their describing functions ($N(A, \omega)$), and the linear part by its frequency response function ($GH(j\omega)$). Then, the solution of the Equation (4.7) yields the amplitudes and frequencies of the limit cycle (Gelb & Velde 1968).

$$1 + N(A, \omega)GH(j\omega) = 0 \tag{4.7}$$

In this case, the linear part is given by Equation (4.2), and the dead zone combined with the saturation have the describing function shown in (4.8), where A is the amplitude

of the limit cycle after the P controller, and δ_1 and δ_2 are the limits of the dead zone and saturation respectively. The function $f(\gamma)$ is called *saturation function* and its values are given by Equation (4.9).

$$N(A) = f(\frac{\delta_2}{A}) - f(\frac{\delta_1}{A}) \tag{4.8}$$

$$f(\gamma) = \begin{cases} -1 & \text{for} \quad \gamma < -1 \\ \frac{2}{\pi}(\arcsin\gamma + \gamma\sqrt{1-\gamma^2}) & \text{for} \quad |\gamma| \leq 1 \\ 1 & \text{for} \quad \gamma > 1 \end{cases} \tag{4.9}$$

Solving Equation (4.7) and assuming that $\frac{1}{\sqrt{\tau KN}} \approx 0$, some useful approximations for the frequency of the limit cycle (ω) and also for the delay margin (T) are found. These approximations, given by Equation (4.10), can be used for control design purposes.

$$\omega \approx \sqrt{\frac{KN}{\tau}} \qquad T \approx \frac{1}{KN} \tag{4.10}$$

An interesting analysis can also be done by observing the polar plot (see Figure 4.29) of Equation (4.7). In this figure, it can be seen that the limit cycles appear when there is an intersection of $GH(j\omega)$ and $-\frac{1}{N(A)}$. Notice that the curve of $-\frac{1}{N(A)}$ has been distorted to show more clearly both limit cycles. Hence, it is clear that there are no limit cycles if there is no dead time (curve $GH(j\omega)$). However, if there is enough delay (curve $GH_T(j\omega)$), then there will be one or two limit cycles. In the case of two limit cycles, one is unstable and the other one is stable. However, the effect of the unstable cycle limit is constrained by the stable limit cycle.

Observing the polar plot of Figure 4.29, it is possible to conclude that at least one limit cycle will occur in the maximum value of $-\frac{1}{N(A)}$ (i.e. when $A = \delta_2$). The minimum value of the delay required for this situation can be calculated using (4.10). For example, if $k_p = k_c = 1$, $\delta_1 = 0.1$, $\delta_2 = 0.2$ and the full brake is selected as in the previous case, then the minimum delay is aproximatly: $T \approx \frac{1}{10N(0.2)} \approx 0.26s$.

The complete model

In this case, the model considered is shown in Figure 4.25, which is controlled by a P controller. The analysis is quite similar to the previous case, the only difference is the automatic reset mechanism of the integrator. This mechanism allows the emulation of

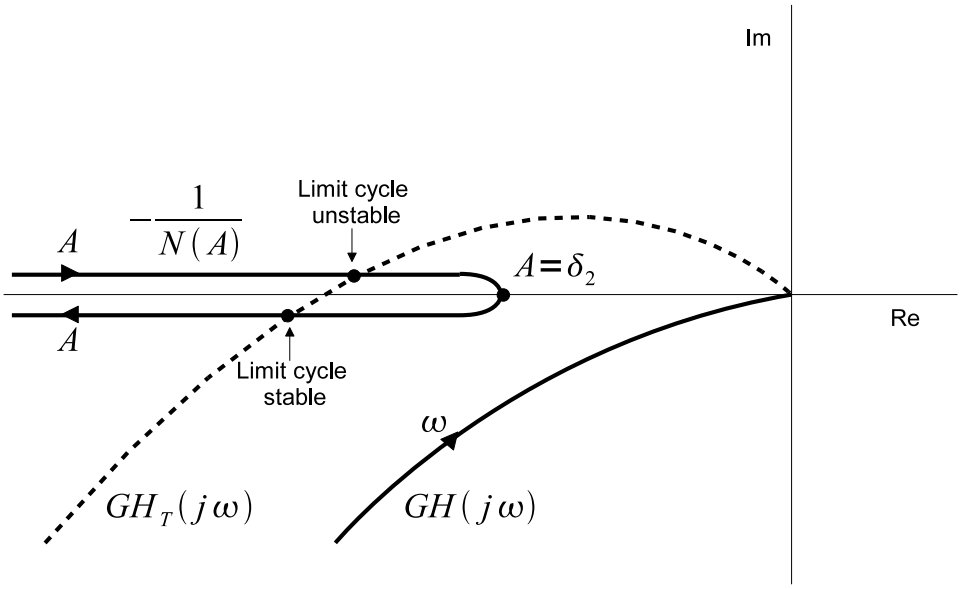


Figure 4.29: Polar plot representation to determine the limit cycle conditions.

the nonlinearity of the position sensor (potentiometer), which keeps the position signal between 0 and 1. For this reason, the behaviour of this model is similar to the previous case when the position signal is ranged from 0 to 1. Otherwise, if the signal crosses these limits then the model could become unstable. Hence, the limit cycles analysis is useful to determine the delay (T) with which the system would have a limit cycle with an amplitude before the P controller is greater than the position range. To do this, the approximation for the time delay given by Equation (4.10) when $A = 0.5k_c$ can be used. So, for instance, if the full brake is selected and $k_p = k_c = 1$, $\delta_1 = 0.1$, $\delta_2 = 0.2$, then the maximum delay is about $T = 0.41s$.

4.4.7 Graphical user interface of the networked control lab

The many control aspects described above demand a graphical user interface (GUI) for the application which offers a high degree of flexibility. But, at the same time, the intended pedagogical use recommends keeping the interaction with the students relatively simple and intuitive. For this reason, a considerable part of the implementation time to discuss and test the possible designs of the interface is needed. The fast-prototyping facilities of EJS were very useful, because they allowed faculty with not much programming expertise to interact directly with the computer at the application design phase, testing the many different possibilities. The result is a modern-looking, intuitive GUI that offers a clear view of the process and allows flexible user interaction without over-

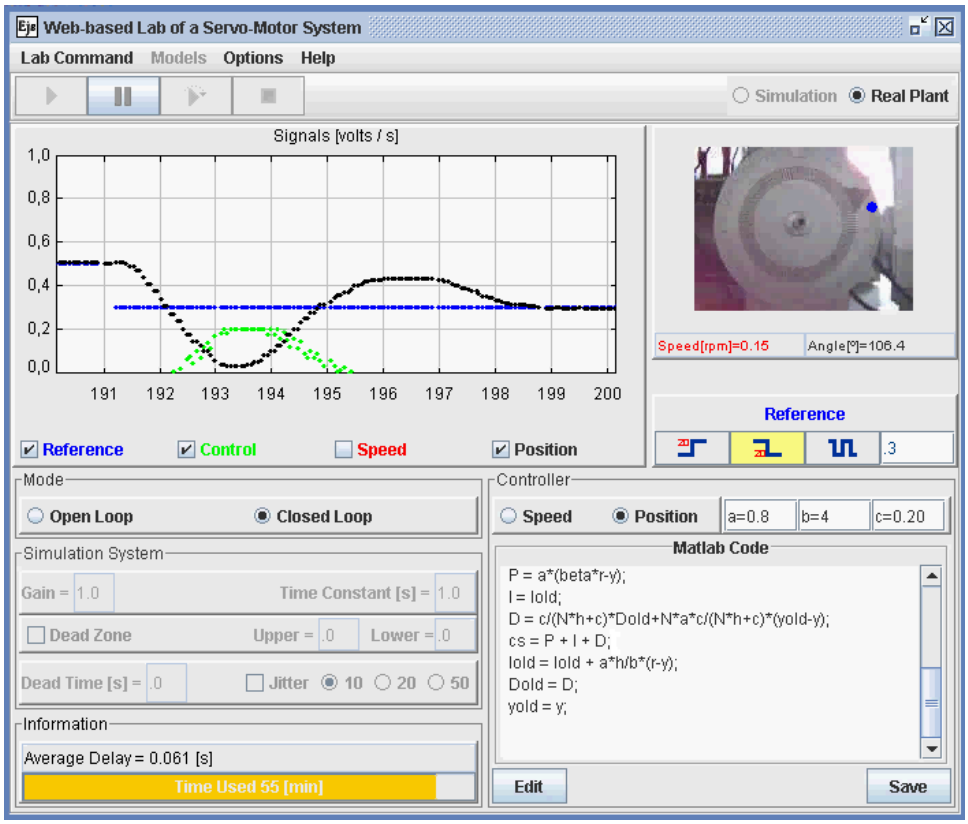


Figure 4.30: Graphical user interface of the implemented remote lab.

whelming the student with too many options.

The interface of the application, displayed in Figure 4.30 has four main panels, a menu bar, and a small task bar.

The two upper panels of the interface provide a quick view of the motor and a time plot of the signals from and to it. The time plots on the left panel display the speed and position signals of the motor, which are read from the JIM connection to the equipment, as well as the reference and the input to the plant (which is the command or the control signal depending on whether the mode is open or closed loop, respectively).

The top part of the right panel shows a real view of the actual motor at Ghent University, obtained by EJS from an IP camera pointing at it. Although the quality of the video thus obtained depends on the speed of the connection to the Internet, the view of the actual equipment brings students a sense of reality. The view includes information on the speed and position of the servo. Also on this right panel, a series of buttons allow students to apply ready-made step up, step down, or square signal as input to the system. The text box allows entering a constant value as input to the plant. In open loop mode, this input represents a command to the plant, while, in closed loop,

the input represents a reference to the controller.

The two lower panels are devoted to the control of the motor. The left bottom panel allows the student to select an open or closed loop mode of operation, as well as modification of the parameters of the controller in closed loop mode. An information bar at the bottom of this panel shows the total delay in each operation cycle. This delay includes the sampling period, the controller computing time, and the round trip communication time between the server and the controller. Finally, a time bar indicates the time elapsed by the current session.

The right bottom panel is used by the student to select if the signal to control is the speed or the position of the motor and to enter the MATLAB code that will be used to compute the control signal. This code accepts three free parameters (`a`, `b`, and `c`) that can be used by the student in the control algorithm (see Listing 4.5) and whose values can be modified using three text fields provided by the interface.

The interface is completed with a top task bar that provides buttons to start, pause, step, and stop the application and select whether the user wants to control the real equipment (if available) or the virtual simulation of the motor.

The components of the interface described above provide the basic functionality required to operate the application. A menu bar provides some additional features such as the possibility to specify the range of the axes of the plot, indicate if the values of the signals are displayed in volts or as percentages, send to the Windows clipboard the values of the signals for further analysis, and calibrate the location of the video of the IP camera.

4.4.8 Using and evaluating the networked control lab

From a wide set of tasks to perform with the networked control lab, the students were asked to do the following activities:

- Control the plant indicating suitable values for the controller (by default a PID controller).
- Apply a step to the motor input voltage.
- Identify the main time constant and the gain from the speed response.
- On the basis of this identification exercise: derive a model for the position response.

- On the basis of the position model: design a controller transfer function for position control.
- Apply the position controller to the motor and evaluate the result.

This exercise was done in a basic course on control engineering in Ghent University, taken by some 130 students. At the end of the use of the networked control lab, the students were invited to evaluate it using an on-line poll (which allows students to vote in a confidential way). The results, from teacher and students points of views, were very good, specially considering it was the first experience. The on-line questionnaire, based on the poll described in (Dormido et al. 2008), were rated for students as *strongly agree*, *agree*, *neutral*, *disagree* and *strongly disagree*. The questions were divided in the following categories:

- Learning Value: The lab helps to learn the relevant contents.
- Value Added: The lab has advantages over traditional learning methods.
- Usability: The lab was easy to use.
- Technical Functionality: The lab works well from software and hardware points of view.

The results indicate that **Technical Functionality** was not a problem for most of students because only 11% evaluated this category as disagree. Regarding the **Usability**, about 84% of students did not find any difficulty in using the lab. About 55% of the students think that the lab helps to understand the matters (**Learning Value**), while only 13% disagree. Regarding the **Value Added**, 30% of the students think that the lab has advantages over other learning methods. However, about 25% of the students think the opposite. Probably, this item indicates the value that students give to the traditional practice in the labs, which means that the remote or virtual labs have to be considered a complement (but not a replacement) of the control engineering teaching.

4.5 Conclusions

After describing how to implement Java classes to support the communication protocol with different well-known engineering software tools, this chapter focuses on provid-

ing authoring tools in order to help to engineering instructors to develop interactive engineering simulations.

The chapter starts with an introduction to the authoring software tool Easy Java Simulations. This software eases the creation of interactive simulations in Java, which can be deployed as standalone applications or embedded applications (i.e. applets) on web pages.

Because the library **JIMC** is a Java package, it can be used by Easy Java Simulations in order to create interactive engineering simulations with MATLAB/Simulink. Although this way to add interactive human interfaces to MATLAB/Simulink simulations is quite direct, the process of linking client and external variables could be cumbersome, specially when Simulink models are used. For that reason, the functionality to connect to engineering software was embedded into a new version of Easy Java Simulations. Thus, instructors can use this new version of EJS to easily set the client and external variables through the Variables subpanel. Other actions, such as connecting to external application, can be accessed by authors using the object `_external`. These features have been implemented for MATLAB, Simulink, Sysquake, and Scilab.

Finally the chapter shows the development of remote laboratories using the interoperate approach. Concretely, the implementation of a networked control laboratory is described. Using this laboratory, students have to consider network delays as part of the control problem. Students can also implement and test on-the-fly their own control algorithm using any MATLAB toolbox. This laboratory has been used in an introductory course of engineering control at Ghent University with positive feedback from students.

Chapter 5

Virtual Laboratories of Embedded Control Systems

In this chapter some of the ideas previously shown are used to build virtual laboratories (labs for short) in the particular field of control engineering called *Embedded Control Systems*. An embedded control system consists of the use of a computer whose main task is to apply a control algorithm in order to keep a signal from a piece of equipment or a process within prescribed safety margins, despite disturbances. The control task typically executes periodically and under limited implementation resources (CPU time, communication bandwidth, energy, memory, etc.). If the limited resource is the CPU time, then the system is generically called a *real-time system*.

Real-time systems and control theory both have a long, but separated, tradition. Focus on research of real-time scheduling has been extensive, but very little of this work has focused on control tasks. On the other hand, digital control theory has not addressed the problem of shared and limited resources in the computing system. Instead, it is commonly assumed that the controller executes as a single loop on a dedicated computer (Dong-Jin 2006). This misunderstanding has normally implied wrong assumptions such as considering that the computation delay of the controller is fixed or the controller deadline is always critical. Nevertheless, many control algorithms have a varying computation time (model predictive controllers), and a single missed deadline does not necessarily cause system failure.

Nowadays, a new interdisciplinary approach is emerging where control and real-time issues are discussed for each of the two design levels. The development of algorithms for co-design of control and real-time systems requires new tools. One of these new tools is TrueTime, which is a freeware MATLAB-based simulator for networked and embedded

control systems that has been developed at Lund University since 1999.

However, as seen before, simulations of Simulink models lack the interactivity and visualization required to learn in a natural way. Without these aspects, simulations of embedded control systems can be hard-to-understand learning objects. Here is where all the work shown in previous chapters comes in handy. Two approaches to create virtual laboratories (labs for short) for embedded control systems are presented in this chapter.

The first approach (introduced in (Farias, Årzén, Cervin, Dormido & Esquembre 2009, Farias et al. 2007)) combines the features of TrueTime and Java in order to build simulations with a rich level of interaction and visualization. Since this approach limits its use to MATLAB users to carry out the simulations, a second approach (introduced in (Farias, Cervin, Årzén, Dormido & Esquembre 2008, 2009)) is presented in order to make the study of embedded control systems possible for a wider audience.

The second approach presents an open source Java library, which is called **JTT** (Java TrueTime) (Department of Computer Science and Automatic Control, UNED 2010*c*). The library is addressed to authors mainly to build simulations of embedded control systems with a high degree of interactivity. Additionally, JTT can be used for authors to create soft real-time applications as will be shown at the end of the chapter. To simplify the implementation of the Java library, the basic real-time task model has been adopted from TrueTime. The functionality is so far limited, only the basic methods to describe real-time kernels and tasks have been implemented. However, it is good enough to provide a simplified way to create simulations of real-time systems for pedagogical purposes. Advanced TrueTime simulation features like wired and wireless communication networks are not yet supported in JTT, and therefore authors that require those advanced features still need to use the first approach.

The chapter is organized as follows. In Section 5.1, embedded control systems are presented. Section 5.2 introduces TrueTime. Section 5.3 describes the implementation and use of the first approach. In Section 5.4, the second approach is illustrated. The creation of soft real-time applications is discussed in Section 5.5. Finally, Section 5.6 presents the main conclusions of the chapter.

5.1 Embedded control systems

In an embedded control system, the (usually multiple) tasks are normally executed in what is called *real-time*. A system is said to be real-time if the total correctness of the operation depends not only on its logical correctness, but also on the time in which it is performed (Burns & Wellings 2001). Real-time systems can be classified in two subcategories: *hard* real-time systems, in which the completion of an operation after its deadline may lead to a critical failure of the complete system and *soft* real-time systems, which tolerate such lateness and may respond with decreased service quality (such as a slower reaction time, or longer settling time).

A simple example is that of stabilizing an inverted pendulum by moving its base back and forth (the academic version of the Segway shown in Figure 5.1). Suppose the operation requirements specify that the pendulum must recover its verticality as soon as possible after suffering any moderate perturbation. If the sampling period of the vertical angle of the pendulum is 80ms, with a time delay of 20ms for the engines to act on the base, a reasonable design could require that the control algorithm be executed every 80ms and have a worst case execution time of 60ms. For the system to avoid the pendulum from falling, the control algorithm must be both correctly designed and applied on time.

5.1.1 Parameters of a real-time task

Real-time tasks such as the control of verticality can be periodic, aperiodic, or sporadic, and are characterized by different parameters, among which are the:

- **release time**: which indicates the next instant in time when a task should be executed.
- **finish time**: to signal when a task has finished its execution.
- **execution time**: which is the duration of the execution of the task. It can be calculated by subtracting the release time from the finish time.
- **period**: to indicate the amount of time at which a periodic task has to be released. When the task is periodic, the release time is always a multiple of the period.
- **offset**: a delay of the first release time.

Figure 5.1: A Segway Personal Transporter. The inverted pendulum is an academic version of a Segway keeping its verticality.

- **deadline**: which indicates the maximum allowed execution time for a correct execution. It is common to take the period as the deadline of a periodic task.

5.1.2 Scheduling policies

Typically a control task runs in parallel with several other tasks, including other control tasks. This puts focus on the *scheduling policy* of the system, which is the algorithm that decides which task to execute at a given time. The presence of a scheduling policy introduces a new parameter for a task, its *priority* or preference with respect to other tasks in the system.

In the previous example, the control of the pendulum's verticality would typically be a top-priority, periodic task with a period of 80ms and execution time of less than 60ms, which makes a deadline of 80ms reasonable. In cases where there are more tasks competing for CPU resources, a shorter deadline could be prescribed.

Under a scheduling policy, tasks may be in one of the three following states: *running*, *preempted* (or blocked) and *sleeping*. Running means that the task is actually executing. Preempted means that the task needs to be executed, but it is not being executed because another task is running (usually one with higher priority). Sleeping indicates that the task has finished and is waiting for its next release time. A schedule plot such as the one

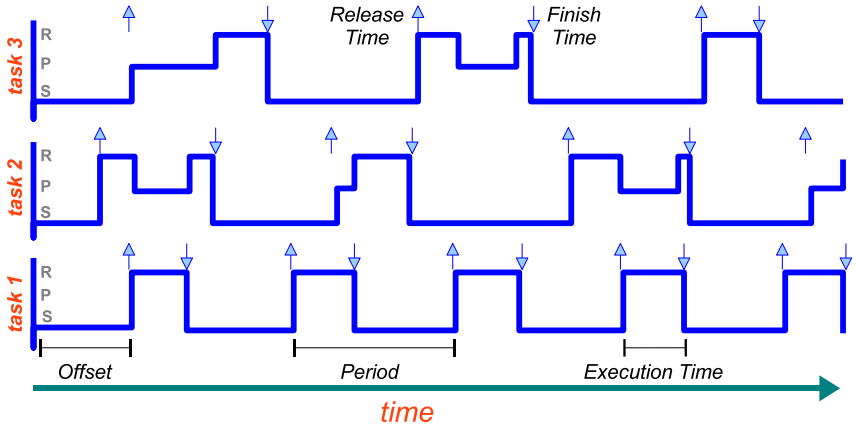


Figure 5.2: Schedule plot: three periodic tasks are running on the same CPU, tasks 1 and 3 have the highest and lowest priority respectively. Up arrows in a task plot indicate the released times of that task, down arrows indicate the task finish times. The initials R, P, and S indicate the possible states of the tasks (i.e., Running, Pre-empted and Sleeping respectively). Note that task 1 is never pre-empted.

shown in Figure 5.2 is a graphical tool used to illustrate the evolution of the states of the tasks in time. Note that task 1 and task 3 have the highest and the lowest priority respectively, which implies that task 1 is never pre-empted and that task 3 is almost always interrupted by task 1 and task 2.

The scheduling policy can be static or dynamic. Normally, a static policy sets the priorities of the tasks before they are executed. On the contrary, a dynamic policy can modify the priority of the tasks while they are running.

Fixed Priority (**FP**) is the simplest static scheduling policy, which assigns the priorities of the tasks arbitrarily. Rate Monotonic (**RM**) is another popular static scheduling policy, which assign the priorities of the tasks on the basis of their period: the shorter the period is, the higher the priority of the task is.

Regarding the dynamic policies, a very well known one is the *Earliest Deadline First* (**EDF**) policy, which places tasks in a priority queue. Whenever a **scheduling event** occurs (a task finishes, a new task is released, etc.) the queue is searched and the process closest to its deadline is scheduled for execution.

The selection of the scheduling policy is not straightforward. Normally, dynamic policies offer much more flexibility than static ones. However since dynamic policies are based on more sophisticated algorithms, the analysis required for each task from the control point of view can be more complex than in the static case.

5.1.3 Codesign problem

Academic interest in real-time systems and control theory have followed different ways during last decades. Since the beginning of the 1970s, the focus on research of real-time scheduling has been very large. Nowadays reaches far even into unconventional areas of application on industry (Nolte & Passerone 2009, Buttazzo & Kuo 2009). However very little of this work has focused on control tasks. On the other hand, digital control theory, with its origin in the 1950s, does not address the problem of shared and limited resources in the computing system. Instead, it is commonly assumed that the controller executes as a simple loop in a dedicated computer.

Typically, the control engineer does not know (or care) about what will happen in the implementation phase of the control algorithm. The common assumption is that the computing platform can provide periodic sampling and the computation delay of the controller is either negligible or constant. Reality tends to be far different. Today, processors are built with caches and pipelines, software is divided into several modules, signals need to be communicated through networks, and there is a strong trend towards the use of commercial, off-the-shelf (COTS) hardware and software. These factors contribute to make the time response of the computing platform, which is shared among many tasks, too hard to predict.

On the other side of the problem, the computer engineer who implements the control system can also make wrong assumptions. It is commonly assumed that controllers have a fixed execution-time, that all control loops are periodic, or that controllers deadlines are critical. In reality, many controllers have varying execution time demands (e.g. model predictive controllers), some controllers are not sampled against time (e.g. combustion engines controllers), and a single missed deadline does not necessarily cause system failure. This misunderstanding between both types of engineers is now been addressed by an emerging interdisciplinary approach, where control and real-time issues are discussed at each design level.

Successful development of an embedded control system requires then a *codesign* of the computer system and the control system. The computing platform must be dimensioned such that all functionality can be accommodated, and the controllers must be designed taking the hardware limitations into account. The codesign problem can be stated as: “given a set of processes to be controlled and a computer with limited computational

resources, design a set of controller and schedule them as real-time task such that the overall control performance is optimized" (Cervin 2003, Cervin et al. 2003).

5.2 TrueTime

TrueTime is a MATLAB/Simulink based simulator for networked and embedded control systems that has been developed at Lund University since 1999 (Lund University 2010). The simulator software consists of a Simulink block library (see Figure 5.3) and a collection of MEX files.

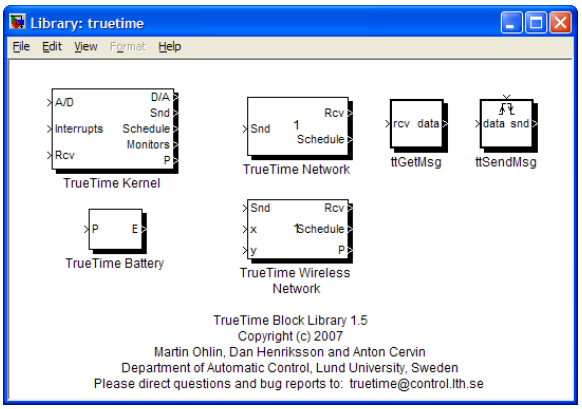


Figure 5.3: The TrueTime 1.5 block library.

TrueTime provides mainly two kinds of blocks, TrueTime Kernel and TrueTime Network. To create real-time simulations, the plant dynamics are first modelled using ordinary Simulink blocks. Then, the computer controlled system is formed by connecting the inputs and outputs of the Kernels and Networks blocks to the model (see Figure 5.4).

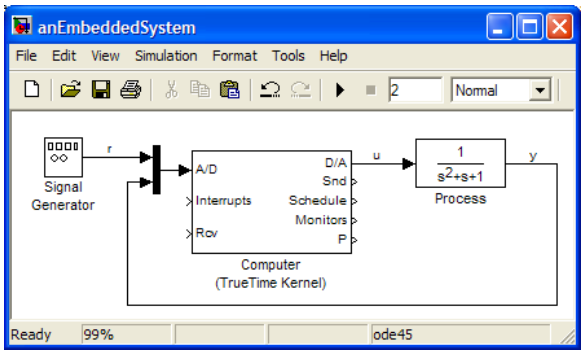


Figure 5.4: A TrueTime simulation of a computer controlled system.

The TrueTime Kernel block simulates a computer node with a generic real-time kernel, A/D and D/A converters, and network interfaces (Andersson et al. 2005). The block is configured via a script (an M-file). The script may be parametrized and the pro-

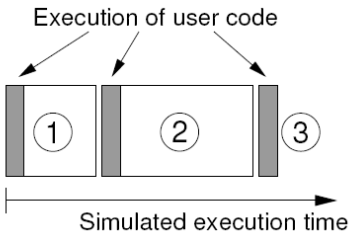


Figure 5.5: TrueTime code model. The execution of user code is modelled by a sequence of segments that are executed in sequence by the kernel.

grammer may create objects such as tasks, timers, interrupt handlers, etc., representing the software executing in the computer node.

Listing 5.1 shows a typical script to initiate the kernel block. The script uses the primitive `ttInitKernel` to configure the kernel block. In this case, the kernel has two inputs (for reference and output signals) and one output (for control signal). The keyword `'prioFP'` informs the kernel that Fixed Priority is selected as the scheduling policy. The primitive `ttCreatePeriodicTask` is used to create a periodic task. The parameters indicate respectively the name of the task (`'controller'`), the offset (`0`), the period (`0.006`), the priority (`1`), the name of the code function to be executed by the task (`'PID'`), and a local variable used as local memory called `data`.

```
1  function initscript
2   % Local data
3   data.u=0;
4
5   % Initialize TrueTime kernel
6   ttInitKernel(2, 1, 'prioFP');
7
8   % Create a periodic task
9   ttCreatePeriodicTask('controller', 0, 0.006, 1, 'PID', data);
```

Listing 5.1: A typical initialization function.

To model the execution time of a task or interrupt handler, a special code function format is used. A code function is divided into code segments according to Figure 5.5. The execution of user code is done nonpreemptively of the beginning of each segment, and the code function returns the simulated execution time of the segment. Within code functions, the user can access the kernel block inputs and outputs using special kernel primitives (e.g. `ttAnalogIn` and `ttAnalogOut`).

Listing 5.2 shows the code function used by the task `'controller'` created in Listing 5.1. Note that a *switch-case* is used to divide the code function into segments (three in this case). The value of the variable `seg` is automatically incremented by TrueTime

in order to execute the code sequentially. The first output of a code function (`exectime` in this case) informs TrueTime of the time used to compute a given code segment. A negative value of this parameters indicates that the execution of the task has finished.

```
function [exectime, data] = PID(seg, data)
 switch seg,
  case 1,
   r = ttAnalogIn(1); % Read reference (first input)
   y = ttAnalogIn(2); % Read process output (second input)
   exectime = 0.001;
  case 2,
   data.u = controlAction(r, y); % Compute control action
   exectime = 0.003;
  case 3,
   ttAnalogOut(1, data.u); % Send out control action
   exectime = -1;
 end
```

Listing 5.2: A typical control task code.

The TrueTime Kernel block supports various preemptive scheduling algorithms such as Fixed Priority (**FP**) scheduling and Earliest Deadline First (**EDF**) scheduling. It is also possible to specify a custom scheduling policy.

The TrueTime Network block and the TrueTimeWireless Network block simulate the physical layer and the medium-access layer of various local-area networks. The types of networks supported are CSMA/CD (Ethernet), CSMA/AMP (CAN), Round Robin (Token Bus), FDMA, TDMA (TTP), Switched Ethernet, WLAN (802.11b), and ZigBee (802.15.4). The blocks only simulate the medium access (the scheduling), possible collisions or interference, and the point-to-point/ broadcast transmissions.

For more details about the use of this toolbox, see the references (Cervin 2003, Cervin et al. 2003, M. Ohlin & Cervin 2007). TrueTime can be downloaded from:

`http://www.control.lth.se/truetime`

5.3 Virtual Labs Using TrueTime

This approach is divided in two phases that use the best features of MATLAB and Java. First, authors design the embedded control system using TrueTime, defining the scheme of the system, the plant to be controlled, the tasks code (e.g. the controller), the tasks features (e.g. periods and priorities), the schedule policy, the network (if needed), etc. Second, authors use Java to control the execution of the simulation by using the JIMC package, and also to build the Graphical User Interface (GUI) of the virtual lab.

Here, instructors can take advantage of the built-in feature to connect MATLAB with Easy Java Simulations (EJS) to add an interactive human interface to the TrueTime simulation. Authors that want to use another authoring tool to build the GUI in Java can still do it by just following the procedure described in Chapter 3 to use JIMC, and applying the same integration process described in this Chapter.

Since creation of simulations in TrueTime and EJS has been described previously, the focus in this section will be on the description of the integration process between both tools in order to get a simulation of embedded control systems with a high level of interactivity and visualization.

5.3.1 Connection Process

Since TrueTime simulations are basically MATLAB files and Simulink models, EJS and JIMC can be used to create simulations of embedded control systems. Chapter 4, Section 4.3, describes how to use the link between EJS and MATLAB/Simulink. This integration process is summarized in the following four actions:

1. Set MATLAB/Simulink as an external application.
2. Connect EJS variables with MATLAB/Simulink.
3. Control and access MATLAB/Simulink.
4. Define the visualization and interactivity.

The facilities provided by JIMC to control MATLAB/Simulink can be considered completely in this integration process. These facilities are the methods of JIMC that allow, for example, to read/write MATLAB variables, to evaluate MATLAB commands, and also to simulate a Simulink model.

To build the simulations using this approach, authors just have to follow the previous procedure for connection between EJS and TrueTime. The first two steps, setting the external application and connecting variables, are relatively simple and require only a few mouse clicks. Controlling the execution of MATLAB/Simulink is carried out using the built-in methods of EJS provided by the `_external` object. This step frequently consists of advancing the simulation time of a Simulink model, evaluating MATLAB commands, and reading or writing MATLAB variables. Since there are normally many interesting

variables declared for local use in the M-files of a TrueTime simulation, accessing the MATLAB variables from EJS can require redeclaring them as global variables.

Probably, the last step of the connection procedure, i.e. defining the visualization and the interactivity, will require a big percentage of the total amount of the design time. For this reason, it is recommended that authors first evaluate all the requirements from the pedagogical point of view of the virtual lab before focusing on the technical aspects of the connection process.

5.3.2 Improving performance

The connection procedure described above guarantees that TrueTime simulations can be linked to EJS in a very direct way. However, in order to improve the visualization and performance of the virtual labs, authors have to consider two aspects of TrueTime simulations: *zero crossing evaluations* and *schedule data*.

The first aspect is taken into account because TrueTime models use scheduling algorithms that involve a lot of zero-crossing evaluations, which make the simulations run slowly. To overcome this obstacle, authors have to indicate that the link between EJS and TrueTime models updates the connected variables at fixed time intervals, which will make the simulation run faster and more smoothly. Otherwise, there will be too much exchange of information among EJS and TrueTime, causing unwanted delays in the simulation. The updating time is specified in the first step of the connection procedure.

The second aspect is related to the schedule data. This information is quite relevant in order to do a successful analysis of a real-time system, because it indicates the states (running, sleeping, or waiting) of a task. For this reason, authors have to force EJS in order to get all the samples from schedule to be sure that these signals are shown in the virtual lab (as the plot of Figure 5.2 shows). This has to be done because EJS tries to get MATLAB variables only a given number of times by default. So, authors have to use the built-in method `_external.setWaitForEver(true)`, to force EJS to wait for MATLAB variables (i.e. schedule data) that sooner or later will be available for reading from the MATLAB workspace.

5.3.3 Examples of virtual labs using first approach

Now, two virtual labs using the TrueTime-EJS integration will be shown. The first one is about a periodic task controlling a simple system. The aim of this example is

just to show how to use the combination between both tools to create virtual labs. The second virtual lab shows the potential of this approach in order to obtain very interesting simulations of embedded control system with pedagogical purposes.

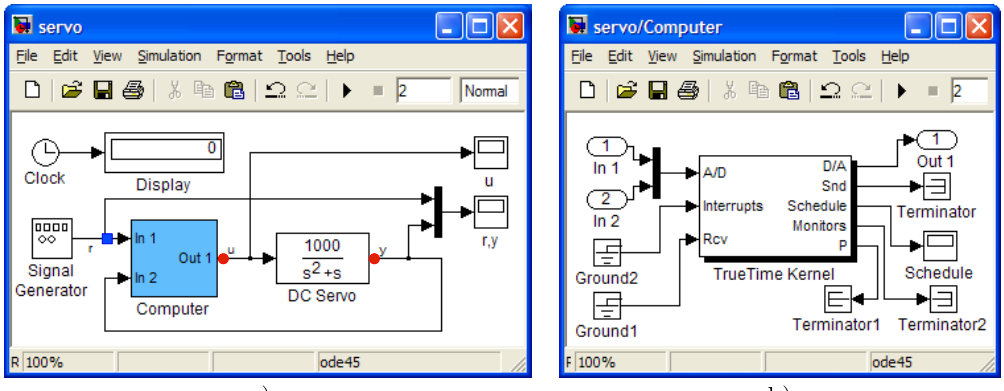
Simple PID servo controller

This virtual lab uses a simple simulation from the list of TrueTime examples (M. Ohlin & Cervin 2007). This example simulates a periodic PID-controller (Åström & Hägglund 2005), embedded in a computer to control a DC-servo process (second order system). The controller is the only task running on the computer. This task is divided into two code segments (see Figure 5.5), one segment to compute the control algorithm, and another one to send out the control action.

The TrueTime simulation uses a Simulink model that represents the complete system, and some M-files to initiate the system and to describe the code function to be executed for the control algorithm. Four different modes of implementation of the task are provided by the example: Built-in Task, Simulink Block, Sleep Until, and Trigger Task.

From the pedagogical point of view, the virtual lab can be used mainly to show how the control performance can be affected by the computing time of the control algorithm (Cervin et al. 2003). In addition to this key concept, the virtual lab should allow students to specify the mode of implementation of the task, to modify the PID parameters, to change the reference, to view the output and control signals, and also to control the period and the computing time of the control algorithm.

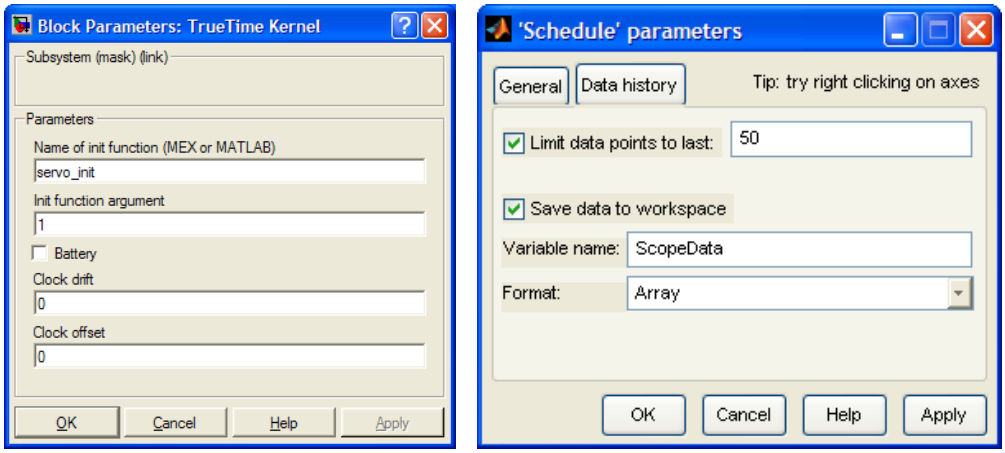
The Simulink model is shown in Figure 5.6a. Note that the feedback control is



a) b)

Figure 5.6: An example of TrueTime simulation. a) Simulink model, b) Submodel of a Computer.

done by the submodel named `Computer`, and the DC-servo process is described by a `Transfer Fcn` block. The `Computer` submodel, shown in Figure 5.6b, uses the `TrueTime kernel` block to simulate a computer. The parameters of the `TrueTime kernel` block (see Figure 5.7a) are used to indicate the `initialization function` (an M-file) that initiates the configuration of the computer, and also to specify an argument which represents in this example one of the four modes of implementation of the task. The parameters of the block `schedule` (see Figure 5.7b) have been modified to add the `ScopeData` variable. This modification will save to the MATLAB workspace schedule data from the task after every integration step of the simulation.



a) b)

Figure 5.7: Parameters of the TrueTime kernel block a) and the Schedule block b).

In order to link the TrueTime simulation with EJS, the first step consists of selecting the Simulink model and connecting the signals to EJS variables. The selection of the model is done by entering the text: `<matlab(0.01)> servo.mdl`, which means that the Simulink model `servo.mdl` will be used as an external application, and that the fixed time interval for updating, used to improve the performance of the simulation, is `0.01` (see Table 4.1 in Chapter 4).

There are in total five EJS variables connected. The variables `time` and `mode` are connected to Simulink parameters to get the simulation time and to set the mode of implementation of the task. The variable `reference` is connected to the first input of the `Computer` block to feed from EJS the reference or set point. Finally, the variables `control` and `output` are connected to the output of the blocks `Computer` and `DC-servo` to read the control and servo-output signal, respectively. The input and output signals

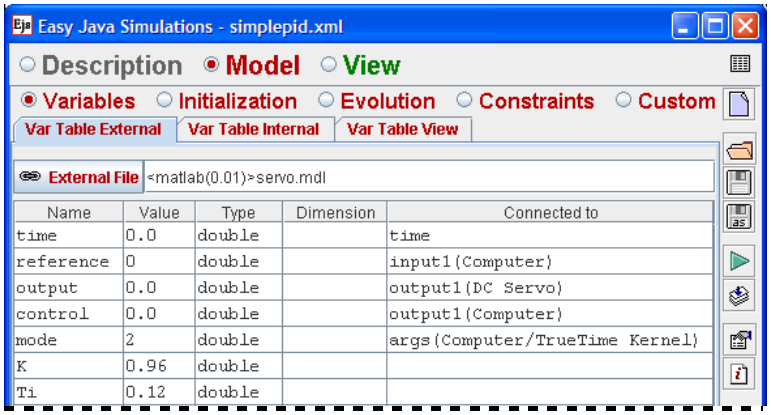


Figure 5.8: Setting a link between EJS and a TrueTime model. Note that a fixed time interval updating is used and the EJS variables are connected to inputs or outputs of Simulinks blocks.

selected are shown as squares and circles, respectively, in Figure 5.6a. The final result of both the selection of the model and the connection of the variables is shown in Figure 5.8.

Before being able to control and to access the MATLAB/Simulink simulation, some modifications of the M-files are needed. The main modification is to adapt the code functions for accessing MATLAB variables from EJS. Since the M-files of TrueTime simulations are mainly functions, a simple way to access the local variables is to redeclare them as global variables.

The first M-file thus modified is the `initialization function` (see Listing 5.3). This M-file is used to initialize the computer where the controller (the task) is executed. The script is divided in two parts, the *initialization* code and the *switch* code.

The initialization code comprises lines 1 to 21. This code uses `ttInitKernel(2, 1, 'prioFP')` to configure a computer with two inputs (reference and DC-servo output), one output (control signal), and a Fixed Priority policy. The initialization also defines and initiates some variables. Note that the variables `period` and `data` are declared as global in lines 8-9 in order to allow access to them from EJS. The variable `period` indicates the period of the task and the variable `data` is used by the task as local memory to save parameters such as the gain, integral time, and derivative time of the PID controller.

The switch code is written in lines 22 to 30. This code executes one of the four different modes to implement the periodic task. Obviously, the selected implementation depends on the input argument `mode` of the `initialization function`. The first mode of implementation is shown in lines 23-26, here the TrueTime function

```
1  function servo_init(mode)
2  % Initialize TrueTime kernel
3
4  % nbrOfInputs, nbrOfOutputs, fixed priority
5  ttInitKernel(2, 1, 'prioFP');
6
7  % Link To EJS
8  global period;
9  global data;
10
11 % Task attributes
12 deadline = period;
13 offset = 0.0;
14 prio = 1;
15
16 % Create task data (local memory)
17 data.K = 0.96;
18 data.Ti = 0.12;
19 data.Td = 0.049;
20 ...
21
22 switch mode,
23   case 1, % IMPLEMENTATION 1
24    % using the built-in support for periodic tasks
25    ttCreatePeriodicTask('pid_task', offset, period,
26      prio, 'pidcode1', data);
27   case 2, % IMPLEMENTATION 2
28 ...
29
30 end
```

Listing 5.3: initialization function.

`ttCreatePeriodicTask` is used to create the periodic task. Note that the M-file `pidcode1` is the code function to be executed for the computer in this mode. Every mode of implementation has an associated code function, but here the focus will be only on `pidcode1` because the modifications are quite similar in the other files.

The script of the M-file `pidcode1` is presented in Listing 5.4. This code can be analyzed in two parts, the *declaration* section and the *code segment* section. The first section, lines 1-4, has been modified to redefine the output of the function and to redeclare the variables `data` and `exectime` as global so that they can be accessed from EJS. The auxiliary variable `exectimeAux` is used to inform TrueTime of the simulated computing time of a code segment. The second section, lines 6-19, describe the two code segments of the task. The first code segment is used to compute the control algorithm and the second one to send out the control signal. Note that the M-file `pidcalc` is called (line 10) to compute the PID control algorithm (Åström & Hägglund 2005). Note also that only the execution time of the first code segment can be modified from EJS using the global variable `exectime`, because in the second code segment the variable `exectimeAux` is equal to a fixed value. As mentioned before, the negative value of the

execution time means that the code segment is the last one of the task and its computing time is zero. Observing this modified Listing 5.4, it becomes clear that EJS end users will be able to modify parameters such as gain, derivative time, integral time, and execution time of the first code segment of the control task.

```
1  function [exectimeAux,data]=pidcode1(seg,data)
2  % function [exectime,data]=pidcode1(seg,data)
3
4  global data exectime; % EJS
5
6  switch seg,
7  case 1,
8   r = ttAnalogIn(data.rChan); % Read reference
9   y = ttAnalogIn(data.yChan); % Read process output
10  data = pidcalc(data, r, y); % Calculate PID action
11  % exectime = 0.002;
12  exectimeAux=exectime;
13
14 case 2,
15  ttAnalogOut(data.uChan, data.u); % Control Signal
16  % exectime = -1;
17  exectimeAux = -1;
18
19 end
```

Listing 5.4: Code function modified to set a link between EJS and TrueTime. Commented lines are original lines of the function.

After the modification of the M-files, authors have to go back to EJS for controlling and accessing the MATLAB/Simulink simulation. As mentioned in Chapter 4, EJS simulations have two main parts (or panels): the `Model` and the `View`. In the `Model` authors describe the behaviour of the system and in the `View` authors use the visual elements provided by EJS to build the GUI of the virtual lab. In the `Model`, there are five subpanels that help authors to systematize the system description process. These five subpanels (see Figure 5.8) are: `Variables`, `Initialization`, `Evolution`, `Constraints` and `Custom`. In this case, only the first three subpanels will be analysed since there is no code in the other ones.

The `Variables` subpanel is used to define the EJS variables. Here, the main variables of the virtual lab were already defined when the link between EJS and the TrueTime simulations was established (see Figure 5.8).

The `Initialization` subpanel is normally used to prepare the simulation before it runs. Therefore, in this virtual lab, an initialization page is used to initiate TrueTime, to execute some MATLAB commands, and to set the initial values of some variables (see Listing 5.5). Note that the function `_external.setWaitForEver(true)` is used

here to make sure that all variables will be read in every simulation step from MATLAB workspace.

```
1  //Initiate TrueTime
2  _external.eval("addpath([getenv('TTKERNEL')])");
3  _external.eval("init_truetime;");
4
5  //Wait to recover MATLAB variables
6  _external.setWaitForEver(true);
7
8  //Declare Global Variables
9  _external.eval("global period");
10 _external.eval("global data");
11 _external.eval("global exectime");
12
13 //Set initial values
14 _external.setValue("exectime",0.002);
15 _external.setValue("period",0.012)
16 _external.eval("data.K="+0.96);
17 _external.eval("data.Ti="+0.12);
18 ...
```

Listing 5.5: Initialization code in EJS.

In the `Evolution` subpanel the actions to be executed by EJS in every simulation step are described. The code used in the virtual lab is shown in Listing 5.6. Two important actions are required in this virtual lab: executing the Simulink model and getting the schedule data.

The first action involves using the built-in method `_external.step(int n)`. The effects of this method, as mentioned in Chapter 4, are: to send to MATLAB the values of the connected EJS variables, to run the Simulink model as many steps as the parameter in parentheses states, and to retrieve from MATLAB the values of all connected EJS variables.

The second action gets the values of some particular variables like `ScopeData`, which is updated by the `Schedule` block (see Figure 5.7b) after each simulation step. To get this data, the built-in method `_external.getDoubleArray()` is used. Note that the information is retrieved separately in two EJS variables (two arrays), `scopeT` and `scopeS`. This is because a `Polygon` was used as a visual element in the view of the virtual lab to properly show the schedule data.

The final action of the combination of TrueTime-EJS approach is about the visualization and interactivity. To create the graphical user interface the visual elements provided by EJS were used. These elements are available in the `View` of EJS (see Figure 5.5). With these visual elements, the user interface of the virtual lab, as shown in

```
1  ...
2  // Stepping the Simulink model
3  _external.step(1);
4
5  // Getting the Scheduler Data
6  _external.eval("scheT=ScopeData(end-49:end,1)'");
7  _external.eval("scheS=ScopeData(end-49:end,2)'");
8  scopeT=_external.getDoubleArray("scheT");
9  scopeS=_external.getDoubleArray("scheS");
10 ...
```

Listing 5.6: Evolution code in EJS.

Figure 5.9, was created.

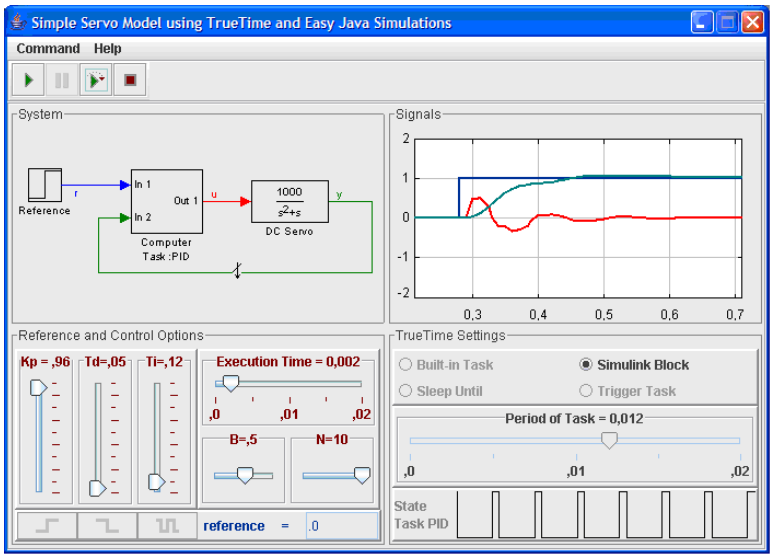


Figure 5.9: Graphical User Interface of the first example. Note that it is possible to change the control parameters on-the-fly using the sliders.

To add interactivity to the virtual lab, it is necessary to define what will happen when end users interact with the visual elements. For instance, note that the sliders are used to manipulate the PID parameters (`kp`, `ti` and `td`) and the *execution time* of the controller. Sliders allow end users to change the values of those variables while the simulation is running easily and quickly. An example about how to add interactivity can be observed in Figure 5.10, which corresponds to the parameters of the slider that controls the execution time of the task (actually, it only changes the execution time of the first code segment of the controller).

The interaction is added to the visual elements just by indicating what to do when the slider is *pressed*, *dragged*, or *released*. For instance, in this virtual lab, if the slider that controls the execution time is moved and released by the user, then the value of the MATLAB variable `exectime` has to be updated to the new value using the following

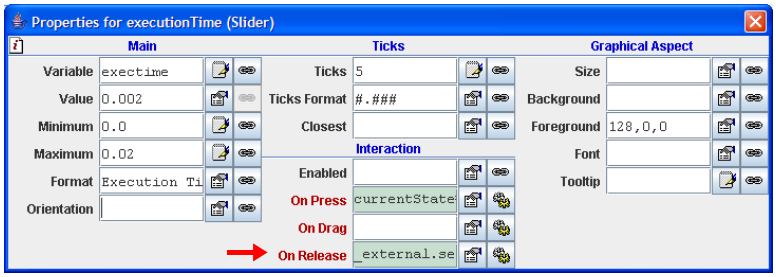


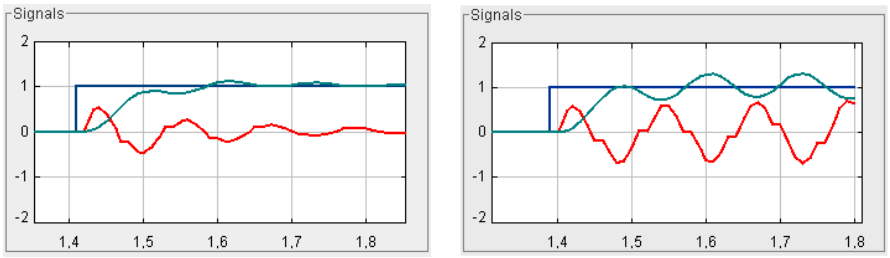
Figure 5.10: Parameters of the slider that controls the execution time of the task.

sentence:

```
_external.setValue("exectime",exectime);.
```

This action also has to be invoked by the rest of the sliders that control the other parameters of the task.

The virtual lab created allows end users to modify a great number of parameters of the system, such as the reference type, control settings, execution time of the controller, and the mode of the implementation of the tasks, etc. As an example of interaction, Figure 5.11 shows the performance of the controller when the execution time is 5 ms or 9 ms. In both cases the Period is 12 ms and the control parameters are the same. However, the control performance of the first case is better than the second one.



a) b)

Figure 5.11: Reference, control, and output signals when the execution time is a) 5 ms and b) 9 ms.

Distributed Servo Control

This example simulates a distributed control of a DC-servo (M. Ohlin & Cervin 2007). The example contains four computer nodes, each one represented by a TrueTime kernel block, connected by a network (see Figure 5.12). A time-driven sensor node samples the process periodically and sends the samples over the network to the controller node. The control task in this node calculates the control signal and sends the result to the actuator node, where it is subsequently actuated. The simulation also involves an interfering node

sending disturbing traffic over the network, and a disturbing high-priority task being executed on the controller node.

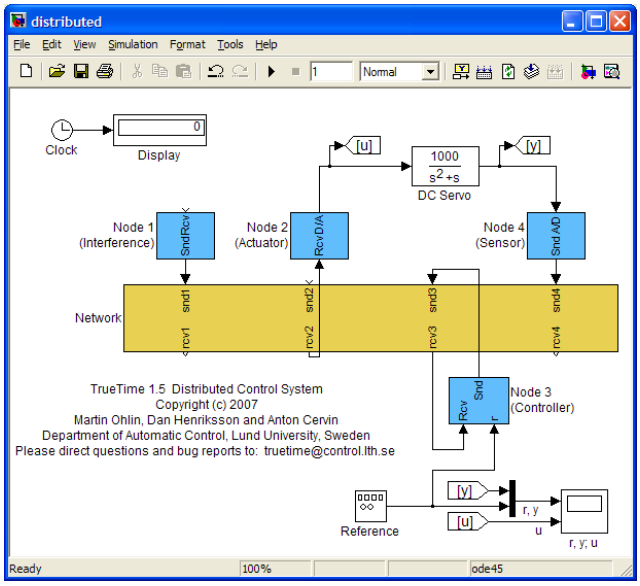


Figure 5.12: Simulink model for the distributed servo control.

This system is a bit more complex than the first example. The main difference is that here there are more M-files and also a bigger Simulink model to modify. However, the connection process to create a virtual lab is quite similar to the previous example, and although it is a bit longer, the integration between both tools is still easy to do.

The virtual lab created is shown in Figure 5.13. The main view (Figure 5.13a) allows the end users to modify parameters of the network and nodes. There are also three auxiliary dialogs to show a histogram of the end-to-end latency (Figure 5.13b), to show the schedule data of the four tasks (Figure 5.13c), and to modify the controller code (Figure 5.13d). This last dialog window gives great flexibility to the virtual lab, because end users can test different control strategies to face the effects of the disturbances due, for instance, to the network. Note also that the controller code is written in MATLAB code, which means that end users can use any MATLAB toolbox available on their computers.

In the main view end users can modify control parameters and also add a mock disturbing high-priority task with different execution times. In the same window, but in the section *Sensor*, end users can modify the measurement time and the package size. This last parameter is important to see the effect of the size of the package on the control performance. The section *Interference* allows end users to increase bandwidth

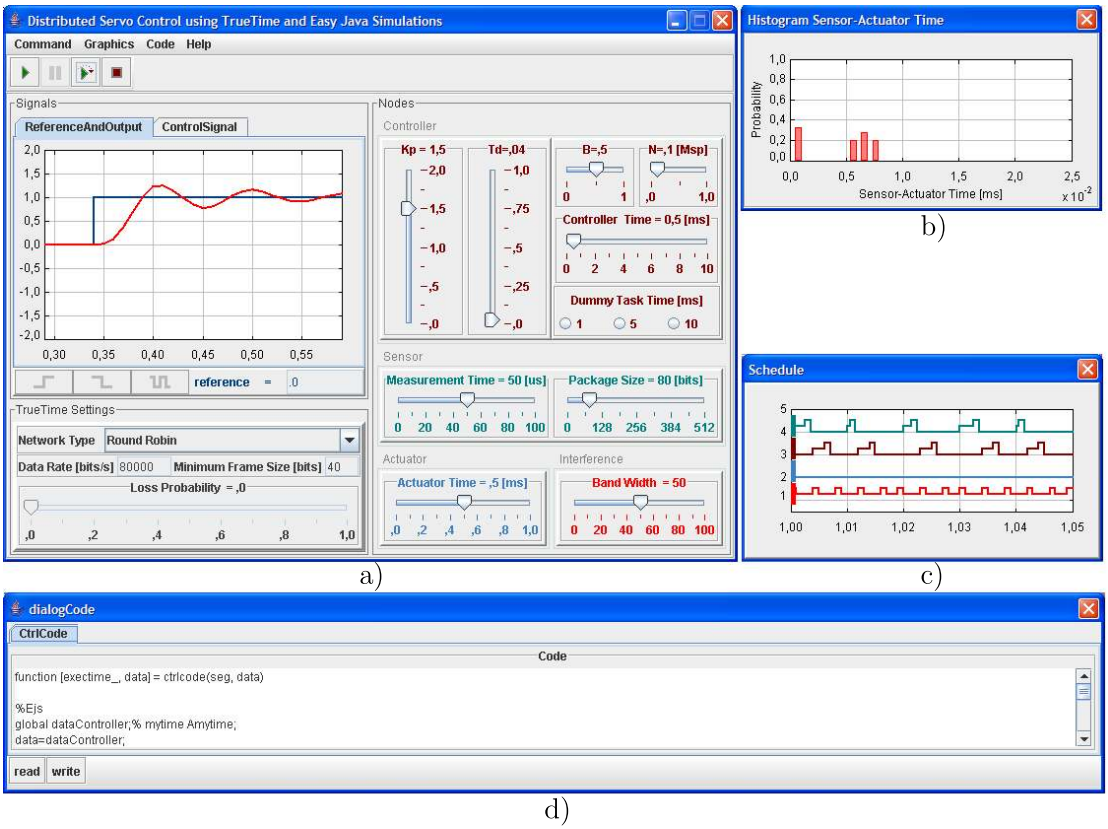


Figure 5.13: Graphical user interface of the distributed servo control. Figures: a) is the main view, b) show histograms of sensor-actuator time, c) is the scheduler data for the four nodes, d) is the dialogCode where the user can read and modify the code of controller.

used by the interference node, which adds some disturbance to the network. The network parameters, such as transmission rate or loss package rate, can be modified in the *TrueTime Settings* section of the main window.

5.4 Virtual labs using JTT

Inspired by the successful ideas and architecture of the TrueTime toolbox for MATLAB, the JTT Java library was implemented in order to simulate code execution and scheduling of tasks in a real time environment. The library allows converting a Java simulation of a control process into an embedded control system by defining one or several kernels (which simulates a computer) that execute tasks according to a given scheduling policy, including RM and EDF. Similarly to TrueTime, the code of a task is divided into segments as shown in Figure 5.5. The kernels, tasks and policies are also highly configurable. Since JTT is a Java library, the systems can be implemented using pure Java code. The programmer creates and adds the system components using the Application Programming Interface (API) of the library, and the library manages

automatically all the internal data structures and organizes the execution of the tasks. Non-programmers can use the library via Easy Java Simulations as Subsection 5.4.5 shows. The JTT library and some examples can be downloaded from:

`http://lab.dia.uned.es/jtt`

5.4.1 JTT's application programming interface

The JTT package provides four public classes and one abstract class. The three most important are: `RTenv`, `Kernel`, and `Task` (see Figure 5.14).

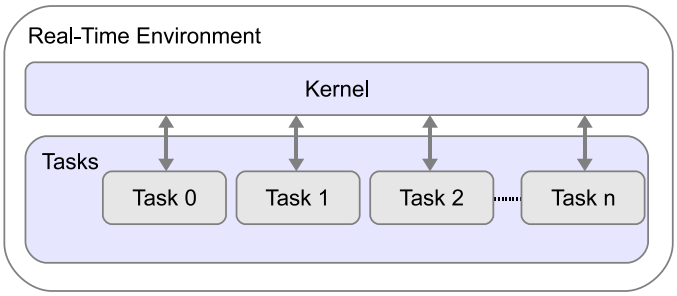


Figure 5.14: Main classes in the JTT package.

A real-time Java environment is an object of the public class `jtt.RTenv`, which provides the basic functionality for implementing the real-time environment. `jtt.RTenv` is a singleton class that can not be instantiated from another class. A kernel is an object of the class `jtt.Kernel` that simulates a computer which can execute one or more tasks. Kernels are instantiated using the constructor:

```
public Kernel();
```

Kernels are added to the real-time environment using the static method:

```
public static boolean RTenv.addKernel(Kernel kernel);
```

A task is obtained by using the public constructor of the *jtt.Task* class:

```
public Task();
```

Tasks can be later customized using standard setter and getter methods. Following TrueTime's code model, tasks in the JTT library are divided into code segments. A code segment is an object, programmed by the user, that extends the *jtt.CodeSegment* abstract class. Segments can also be added or removed from the task after instantiation using convenience task methods.

This object-oriented structure provides a flexible and powerful way to create sophisticated tasks. Alternatively, for simple situations, the task's code can also be defined using reflection. This option can use suitable methods to run the code of the task. To divide the method into code segments, the method can include calls to the static method:

```
public static void RTenv.endSegment(double time);
```

With this information, the kernel object manages two internal queues to control the execution of the tasks. The first queue is sorted by priority and keeps the identifiers of tasks which are ready to be executed by the kernel. The second queue is sorted by release time and keeps the identifiers of tasks which are waiting to be released. The kernel uses this task to determine the release time of the next task and to execute the task segments according to the scheduling policy.

Authors can use this API to modify an existing simulation which they can step in time. They can modify their initialization to create and add the required kernels and tasks, and then the main loop to request the time of the *scheduling event* of the closest task in all kernels. If this time is the closest to the desired step, the program hands over the control of the execution to the kernel to execute the task code.

The classes `Kernel` and `Task` are implemented using Java threads. This implementation choice allows interrupting the execution of a task and restarting it when it is released next. To coordinate the execution of the tasks, each kernel object has an object of the private class `jtt.Token`. When the kernel receives the request to run, it gives its token to the task that is to be released. The task returns the token to the kernel when it finishes the execution of a code segment and the kernel returns the control to the calling program.

5.4.2 Sample implementation

To briefly exemplify this structure, suppose the original program consists of the following, rather simplistic pseudo-code shown in Listing 5.7.

To convert this process into an embedded control system, the programmer needs to modify this class as Listing 5.8 shows. Here, some API methods for creation and configuration of the tasks and kernels were used. Note how the simulation of the embedded process is done in the static method `main`. The next subsection will discuss this in detail.

```
1  public class MyProcess {
2
3   // Initializes the process
4   public MyProcess() {
5     ...
6   }
7
8   // Steps the process for an increment of time
9   public void step(double dt) {
10    ...
11  }
12
13  // Simulate the process
14  static public void main (String[] args) {
15    double time = 0, tFinal = 1.0, dt=0.001;
16    MyProcess process = new MyProcess();
17    while (time<tFinal) {
18      process.step(dt);
19      time += dt;
20      // output process variables
21      ...
22  }
23 } // end of class
```

Listing 5.7: Original simulation.

```
1  import jtt.*; // Import the JTT package
2
3  public class MyProcess {
4
5   // Initialize the process
6   public MyProcess() {
7     ...
8
9     // create a task
10    Task task1 = new Task();
11    task1.setPeriod(0.08); // period 80 ms
12    task1.setPriorityValue(0) //Top priority
13
14    // add code to a task
15    task1.addCode(new CodeSegment(){
16           // code of the first segment
17           public double code(){
18             ...
19             return 0.03; //execution time
20           }
21         });
22    ...
23
24    // create another task
25    Task task2 = new Task();
26    task2.setPeriod(0.1); // period 100ms
27    task2.setPriorityValue(1) // priority
28
29    // add code to another task
30    ...
31
32    // create kernel and add it the tasks
33    Kernel kernel = new Kernel()
34    kernel.setSchedulingPolicy(Kernel.FP);
35    kernel.addTask(task1);
36    kernel.addTask(task2);
37
38    // add the kernel to the real time environment
39    RTenv.addKernel(kernel);
40  }
41
42  // Step the process for an increment of time
43  public void step(double dt) {
```

```
44      ...
45    }
46
47    // Simulate the process
48    static public void main (String[] args) {
49      double time = 0, tFinal = 1.0, dt=0.001;
50      MyProcess process = new MyProcess();
51      while (time<tFinal) {
52        double nextEvent = RTenv.nextEvent();
53        if (nextEvent<time+dt) {
54          process.step(nextEvent-time);
55          RTenv.runKernel();
56          time = nextEvent;
57        }else {
58          process.step(dt);
59          time += dt;
60        }
61        // output process variables
62        ...
63    }
64  } // end of class
```

Listing 5.8: Modified simulation, version 1.

Listing 5.8 uses one way to add code to the tasks, i.e., overriding the class `CodeSegment`. However, as mentioned above, there is another, much simpler way that can be preferred by non-programming authors, called reflection.

The use of reflection provides beginner programmers with an easy way for adding code to a task. Listing 5.9 shows how this is done. Observe that reflection is used because the method `setReflectionContext` was added at the beginning of the initialization. The input parameter of this method is used to define the Java object which implements the code, in this case, the same class `MyProcess`. Adding code to a task is done by using the method `addCode`, where the input parameter sets the name of the Java method that has to be run when the task is executed. Note how the method `endSegment` is used to split the task's code in code segments. The input parameter of `endSegment` is used to return the execution time of a code segment.

Both ways for adding task's code can be used without distinction in most cases. However, reflection may be easier to use for beginner programmers, whereas overriding the class `CodeSegment` is much more useful if the task's code is modified in runtime.

5.4.3 Integration of JTT in advanced simulations

In general, the simulation of an embedded control system consists of two main parts: the computer and the physical system. The first part simulates a computer (i.e., a kernel) where the control task is executing, while the second part simulates the model of the physical system or the process to be controlled. The JTT package allows authors to

```
1  import jtt.*; // Import the JTT package
2  public class MyProcess {
3
4    // Initialize the process
5    public MyProcess() {
6      //Sets reflection
7      RTenv.setReflectionContext(this);
8      ...
9      // create a task
10     ...
11     // add code to a task
12     task.addCode("mycode");
13
14     // create kernel and add it the tasks
15     ...
16   }
17   ...
18   // code of a task
19   public void mycode(){
20     //code of the first segment
21     ...
22     RTenv.endSegment(0.3); //execution time
23     ...
24     //code of the last segment
25     ...
26     RTenv.endSegment(0.1); //execution time
27   }
28   ...
29 } // end of class
```

Listing 5.9: Modified simulation, version 2.

simulate the computer behaviour, but the simulation of the physical system has to be provided by authors, who have to write the required Java code or use other suitable Java packages or tools like EJS.

Normally, physical systems are modelled using Ordinary Differential Equations (ODE). For this reason typical simulators have various ODE solvers (also called numerical or integration methods) to simulate these ODE models. The implementation in Java of a simple ODE solver simulator should not be a difficult task even for a beginner Java programmer. In fact, there is much open source code available on the Internet such as the Open Source Physics project (Christian 2007, 2010). However, writing all the Java code required for creation of highly visual and interactive simulations is a hard (or at least a time consuming) task for some authors.

Consider, for instance, the ODE model given by Equation (5.1). In this system, the derivatives are given by the function $f(t, x)$ and the initial state of the system is represented by x_n. To solve an ODE model means to advance the system from an initial state x_n to a final state x_{n+1}. The experimented reader have surely noted that this action is implemented by the method `step()` in the sample simulation of Listing 5.7.

There are many solvers for ODE models, one of the most popular ODE solver is the classical Runge-Kutta fourth-order method. This algorithm calculates the final state x_{n+1} by means of a weighted average given by Equation (5.2).

$$\dot{x} = f(t, x), \quad x(t_n) = x_n \tag{5.1}$$

$$x_{n+1} \approx x_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \tag{5.2}$$

This approximation is fifth-order accurate in the step size for a single step. The values of k_1, k_2, k_3, and k_4 are calculated by Equation (5.3), which represent the derivative at beginning and middle times. The symbol $\triangle t$ represents the step size, i.e., the interval time between the initial and final time.

$$\begin{aligned} k_1 &= f(x^n)\triangle t \\ k_2 &= f(x^n + k_1/2)\triangle t \\ k_3 &= f(x^n + k_2/2)\triangle t \\ k_4 &= f(x^n + k_3)\triangle t \end{aligned} \tag{5.3}$$

The coordination of both elements, kernel and solver, can be easily done by a simulator by just repeatedly executing the ODE solver and the kernel at specific times, see Figure 5.15.

When both solver and kernel are executed by the simulator, they internally set their

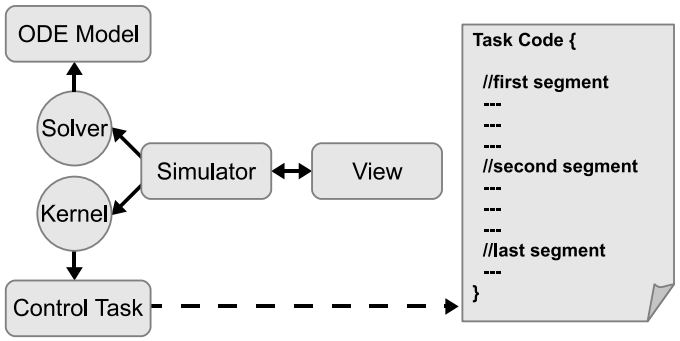


Figure 5.15: Diagram of an embedded control system simulation with JTT. The kernel simulation is provided by JTT. Solver and View have to be programmed or facilitated by other Java tools like EJS.

next time to be called by the simulator. In the case of the solver, this next invocation time is the next integration step determined by the algorithm that implements the solver. In the case of the kernel, the next invocation time is based on the time of the next scheduling event. An event in the kernel can be for instance a task that has finished a code segment, or a task that was sleeping and should be released.

Obviously, the simulator runs the system at the time given by the minimum of both next invocation times. Note that when the kernel has to be invoked, the simulator also has to call the solver in order to get the state of the ODE model at that time. Note also that the control execution of the kernel is done by the primitive `runKernel()`, while the method `nextEvent()` has to be used in order to get the next scheduling event of the kernel. These ideas about the kernel and solver integration are clearly exposed in the static method `main` of the modified simulation described by Listing 5.4.

5.4.4 Using JTT from Java

In this subsection, a virtual lab of an embedded control system is presented. First, the model of a DC servo motor is presented. Then, the Java code to simulate the system is discussed. The objective of this virtual lab is just to show how to create simulations of embedded control systems using Java and the JTT library.

Embedded Control of a DC Servo

In this subsection a DC servo system controlled by an embedded PID controller (Åström & Witternmark 1997) is used. The pedagogical purpose of the virtual lab is to show how the execution time of the controller induces a delay in the feedback loop that might deteriorate the performance.

The embedded system consists of a periodic task controlling a simple DC servo system. The physical system to be controlled is modelled as an ODE given by Equations (5.4) and (5.5).

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1000 \end{bmatrix} u \tag{5.4}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{5.5}$$

The controller is described by a periodic task divided in two subtasks or code segments. The first code segment computes the control action using a PID algorithm. The second code segment takes the computed control action and sends out this signal to the servo system. Since the first subtask consumes much more time from the CPU than the second one, it can be assumed that only the first code segment spends time.

Simulation of the embedded servo in Java

For simplicity, the simulation of the servo motor follows the same structure presented in Listing 5.4. However, two new methods are added to this listing: `getRate(double[] state, double[] rate)` and `step(double dt)`. Part of the new modified simulation is presented in Listing 5.6.

The method `getRate(double[] state, double[] rate)` is shown in line 29 of Listing 5.6. This method is used to describe the servo as an ODE, and also to update the derivatives (or rates) given by Equation (5.4). The ODE model of the DC servo has two states (x_1 and x_2), but, for practical reasons, the time is considered another state of the model. Hence, for convenience, the servo has three states (x_1, x_2 and *time*). The states are coded by the array of doubles `state`, where x_1 is `state[0]`, x_2 is `state[1]`, and *time* is `state[2]`. Note that the rate of the third state (time) is computed in the last rate.

The second method, `step(double dt)`, is added to step the process for an increment of time (`dt`). This method, see line 36 of Listing 5.10, implements the Runge-Kutta algorithm to solve the ODE model of the servo. The method uses the variable `state` and the method `getRate` defined previously.

```
1  import jtt.*; // Import the JTT package
2  public class MyProcess {
3
4   // Initialize the process
5   public MyProcess() {
6    ...
7    // create a task (a PID controller)
8    Task task = new Task();
9    task.setPeriod(0.012); // period = 12 ms
10
11   // add code to task
12   task.addCode(new CodeSegment(){
13     public double code(){
14      controlAction=calculate(reference,output);
15      return 0.02; //20ms
16     }
17   });
18   task.addCode(new CodeSegment(){
19     public double code(){
20          input=controlAction;
```

```
21            return 0; // 0ms
22        }
23      });
24
25      // create kernel and add it the task
26        ...
27    }
28
29    // Gets rate of the ODE model
30    public void getRate(double[] state,double[] rate){
31     rate[0]=state[1];
32     rate[1]=-state[1]+1000*input;
33     rate[2]=1;
34    }
35
36    // Step the process for an increment of time
37    public void step(double dt){
38      getRate(state, rates1);
39      for(int i=0;i<numEqn; i++)
40       k1[i]=state[i]+stepSize*rates1[i]/2.0;
41      getRate(k1, rates2);
42      for(int i=0;i<numEqn; i++)
43       k2[i]=state[i]+stepSize*rates2[i]/2.0;
44      getRate(k2, rates3);
45      for(int i=0;i<numEqn; i++)
46       k3[i]=state[i]+stepSize*rates3[i];
47      getRate(k3, rates4);
48      for(int i=0; i<numEqn; i++)
49       state[i]=state[i]+stepSize*(rates1[i]+
50        2*rates2[i]+2*rates3[i]+rates4[i])/6.0;
51    }
52
53    // Computes the PID
54    public double calculate(r,y){
55      P = Kp*(beta*r-y);
56      I = Iold;
57      D = Td/(N*h+Td)*Dold+N*Kp*Td/(N*h+Td)*(yold-y);
58      Iold = Iold + Kp*h/Ti*(r-y);
59      Dold = D;
60      yold = y;
61      return(P + I + D);
62    }
63
64    // Simulate the process
65    ...
66
67  } // end of class
```

Listing 5.10: Modified simulation, version 3.

The task code of the PID controller is added in the simulation in line 11 of Listing 5.10. In the first segment, the task gets the control action (by calling `calculate`), which is used in the second segment to feed the input of the servo. The variables `reference`, `input`, and `output` represent the reference, the input, and the output of the process, respectively. Note that the code was added by overriding the class `CodeSegment`.

The rest of the simulation is similar to the first modified version (see Listing 5.8). Although this version is quite simple, designed just to show how to use the JTT package in Java, authors with programming skills can use the Java packages `awt` and `swing` to create a more visual and interactive version, adding only a few lines of code to this simulation (see Figure 5.16).

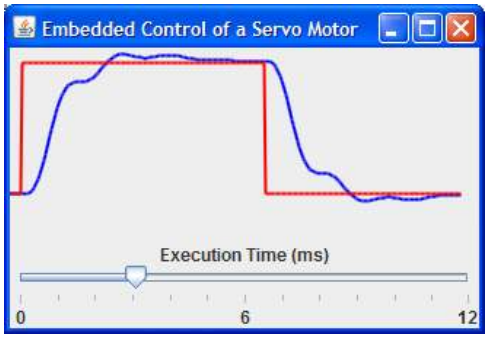


Figure 5.16: A GUI of the embedded servo created by using the JTT-Java approach.

5.4.5 Using JTT from EJS

Here, two examples using JTT library in EJS are presented. The first example is the same simulation of the previous embedded servo system but now using JTT and EJS. This option can be especially useful for authors who prefer to use all the facilities provided by EJS to build simulations with a high level of interaction and visualization. The second example describes the simulation of three inverted pendulums running on one computer.

Simulation of the embedded servo in EJS

As mentioned before, in EJS every application is divided in two main parts (or panels): the `Model` and the `View`. In this approach, the `Model` is used to initialize the embedded system and also to describe the ODE model and the code function of the task. In the `View`, the visual elements of EJS are used to build the GUI of the simulation. Before starting writing code, the JTT package has to be imported and the variables have to be declared. To import the JTT package, EJS provides a special dialog window to browse the file `jtt.jar` and to enter the corresponding import statement (see Section 4.1.1 in Chapter 4). The `Variables` subpanel is used to declare the kernel and tasks variables.

In Listing 5.11 the *Initialize the Embedded System* script is shown. This script is written in the `Initialization` subpanel of EJS. Note that the code for *Initialize the Embedded System* is almost the same as the code shown in previous Listings. However, here it is specified that the schedule data must be available for plotting purposes. This is done using the methods `setSchedule` and `setScheduleWindow`. The second method allows the authors to define the time extension of the schedule data available.

Regarding the task code, it can be seen that the reflection was selected to implement

```
1  //Initialize the Embedded System
2  RTenv.setReflectionContext(this);
3  kernel = new Kernel();
4  kernel.setSchedule(true);
5  kernel.setScheduleWindow(0.5);
6  task = new Task();
7  task.setPeriod(0.012);
8  task.addCode("taskcode");
9  kernel.addTask(task);
10 RTenv.addKernel(kernel);
```

Listing 5.11: Creation of the kernel and task in EJS.

the embedded system. Similarly to Listing 5.9, the method `setReflectionContext` in Listing 5.11 defines that `this` is the object where the code function of the task is located. This should be the general situation in EJS, since methods defined by users, like the code function `taskcode`, are normally located in the section `Custom`. Obviously, the other way to add task's code (i.e., extending the class `CodeSegment`) is also possible in EJS.

Listing 5.12 shows the method that implements the code function of the `taskcode`. Note that `taskcode` is quite similar to the previous version of the simulation (see Listing 5.10). However, since reflection is used here, the task code is divided now into code segments using the method `endSegment`. The input argument of `endSegment` represents the execution time of the code segment previously mentioned. Other user methods like `calculate`, which returns the computed control action, are also implemented in the section `Custom`.

```
1  public void taskcode () {
2   //Update Output
3   output=x1;
4   //Calculate Action
5   controlAction=calculate(reference,output);
6   RTenv.endSegment(executionTime);
7   //Send Out Control Signal
8   input=controlAction;
9   RTenv.endSegment(0);
10  }
```

Listing 5.12: The method "taskcode" used by the periodic task.

In the `Evolution` subpanel of the `Model` all the code that should be executed continuously by the simulation has to be implemented, i.e., the solver and kernel events, and also the plotting of the schedule data.

Regarding the solver and the plant dynamics, the ODE mode given by Equation (5.4) is implemented using the ODE editor provided by EJS (see Fig 5.17). As in the previous

simulation, the Runge-Kutta algorithm is also selected here.

The event detection feature of EJS, by using a bisection method, will be quite useful here for the detection of scheduling events. The events in EJS are added by pressing the `Events` button (see Fig 5.17) and defining two parts. The first part represents the so-called *zero cross function*, which returns zero when the event must be triggered. The second part of the event represents the *action* of the event, which is a set of statements that have to be executed when the event is triggered. Take into account that solvers in EJS always update the state of the ODE model before calling any event.

Using this feature of EJS, the detection of the kernel event is quite simple. The *zero cross function* is just the time remaining into the next scheduling event, i.e., `return RTenv.nextEvent(t);`. The *action* is also simple, because the only statement needed to call the corresponding kernel is `RTenv.runKernel();`.

To capture the schedule data, the code in Listing 5.13 is also added to the `Evolution` subpanel, but in a second evolution page called `getSignals` (see Fig 5.17). This script uses the method `getSchedule` to get schedule data (arrays `time` and `value`) of the task. This data will be used by a `polygon` (a visual element of EJS) to plot the schedule state of the task in the GUI of the simulation.

```
1  ...
2  //*******Capture Schedule Signals******
3  taskSchet=task.getSchedule("time");
4  taskSchev=task.getSchedule("value");
5  points=taskSchev.length;
6  ...
```

Listing 5.13: Getting the schedule data of the task.

After implementing the code in the `Model`, the visual elements provided by EJS are

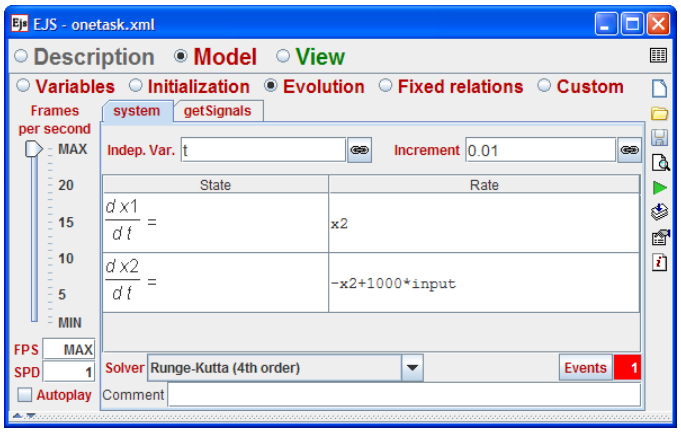


Figure 5.17: Ordinary differential equations of the system using the editor of EJS. The ODE models are defined in the `Evolution` subpanel in EJS.

used to build the GUI of the simulation. Figure 5.18 presents the final result of the use of visual elements of EJS. Four kinds of elements are quite important in this view. The PlottingPanels used to show the axis of coordinates. The Traces: `output`, `reference`, and `control`, used to plot the output, control, and reference signals of the system. The `state` polygon used to graph the schedule data. And finally the `executionTime` slider which allows end users to modify the execution time of the first code segment of the controller (see Listing 5.12).

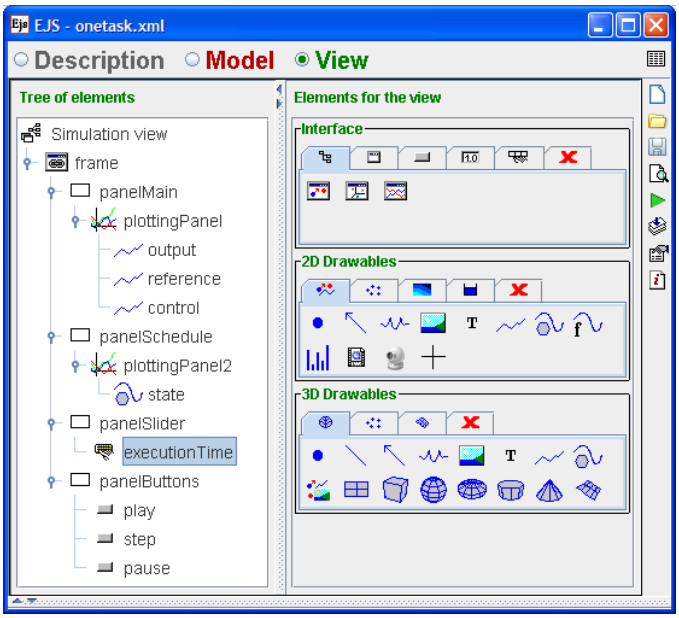


Figure 5.18: View panel of EJS. Elements on the right are provided by EJS to build the tree-like structure on the left, which describes the GUI of the simulation of Figure 5.19.

The GUI of the simulation is shown in Figure 5.19. The virtual lab has two plots. The upper plot shows the signals reference, control, and output of the system. The bottom plot, presents the schedule data of the task. There is also a slider to control the execution time and three buttons to control the simulation. Using this virtual lab, end users can see how the increase of the execution time of the controller negatively affects the control performance. This can be seen in Figure 5.19, where the execution time has been changed from 2ms to 8ms at 1.5s.

Control of Three Inverted Pendulums

This virtual lab simulates a more advanced example than the previous ones. In this case, three inverted pendulums of different lengths should be controlled by a computer with limited computational resources (Cervin 2003). The control objective of the system is

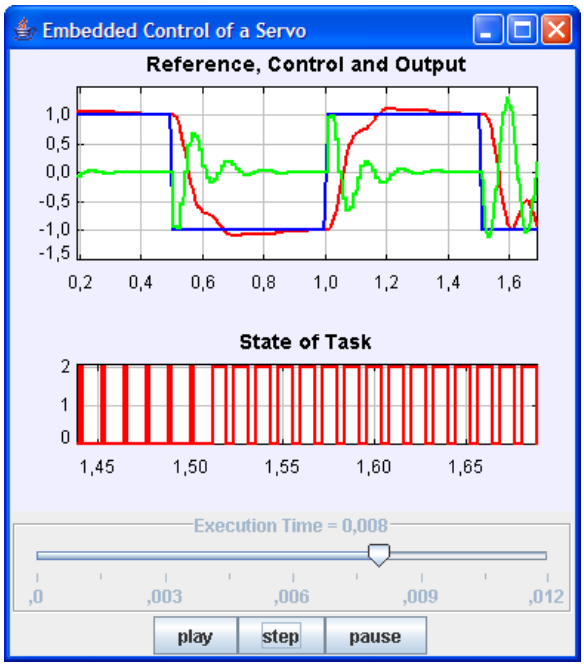


Figure 5.19: GUI of the virtual lab developed using the JTT-EJS approach.

to reach a desired position for the cart, while the pendulum maintains its verticality. A linear digital controller (Åström & Witternmark 1997) is designed (by state-space method) for each pendulum. The pendulum lengths motivate different periods for the three controllers (since it is easier to control longer pendulums). Other parameters, such as control gains or execution times, should be similar in all cases.

The inverted pendulums are modelled by Equations (5.6) and (5.7), see details in (Dorf & Bishop 2004, Ogata 2006). The variables x and $\dot{x}$ represent the position and velocity of the cart, respectively, and the variables θ and $\dot{\theta}$ are the angle and the angular velocity of the pendulum, respectively. The constants M, m, b, l, I are the cart mass, pendulum mass, cart friction, pendulum length, and pendulum inertia. The control objective of the system is to reach a desired position for the cart, while the angle of the pendulum is totally vertical. As in the previous example, the control task is divided in two code segments, one segment to compute the control action. The control action is given by Equation (5.8), where k_1, k_2, k_3, and k_4 are calculated by using the LQG algorithm (Dorf & Bishop 2004, Ogata 2006).

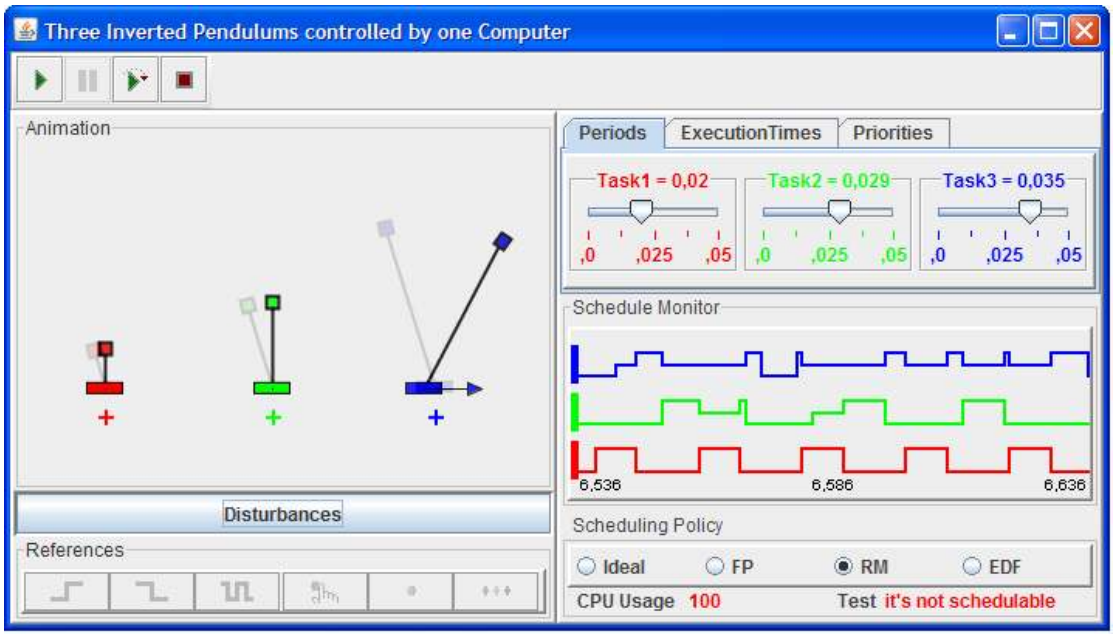


Figure 5.20: A virtual lab built with EJS and JTT. Three inverted pendulums controlled by three periodic controllers running on the same computer.

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-(I+ml^2)b}{q} & \frac{m^2gl^2}{q} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{-ml^2b}{q} & \frac{mgl(M+m)}{q} & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{I+ml^2}{q} \\ 0 \\ \frac{ml}{q} \\ 0 \end{bmatrix} u \tag{5.6}$$

where $q = I(M+m) + Mml^2$

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} u \tag{5.7}$$

$$u = r - k_1 x - k_2 \dot{x} - k_3 \theta - k_4 \dot{\theta} \tag{5.8}$$

The GUI of this virtual lab is shown in Figure 5.20. On the left side, an animation of the three inverted pendulum is presented. Students can select different types of references, manually modify desired position points, and even apply disturbances to the angle of the pendulum. On the right side, users can modify the parameters of all tasks such as period, execution time, and priority. It is also possible to select one of the three

scheduling policies, among other options.

The main goal of this virtual lab, from the pedagogical point of view, is to show how the scheduling policy affects the control performance. For instance, Figure 5.20 shows the state of the pendulums two seconds after a moderate disturbance was applied. Note that two pendulums have totally recovered the verticality; however, the largest pendulum is still trying to become stabilized. This fact is a consequence of the scheduling policy selected. Because the Rate Monotonic (*RM* at the GUI) was selected, the task's priorities are sorted by the period. The largest pendulum has the largest period and hence, the lowest priority. This introduces variable delays in the execution of the controller, due to the interruption of the other two pendulums (see the schedule plot at the GUI). If the scheduling policy is changed to Earliest Deadline First (*EDF* on the GUI), the CPU is shared between the task in a fairer way, and the verticality of all pendulums can be achieved at approximately the same time. More details about this system can be found in (Cervin 2003).

5.5 Soft real-time applications using JTT

JTT has mainly been designed to simulate embedded control systems as shown above. However, there is also another mode of operation that can be used to create soft real-time applications.

In some cases, authors may want to execute their embedded control systems as real applications (possibly interfaced to real plants), and even according to real-time constraints. This is, in principle, not possible, since the JTT library is used by ordinary Java programs without any timing guarantees. However, given a fast enough computer, a Java application may be time-stable over a long execution. Hence, as long as authors do not care about hard time constraints and just want to execute an application (for instance to control a real laboratory process), they can use JTT in soft real-time mode for this purpose.

There are some important differences between the two modes of operation. In simulation mode, authors can use JTT to create as many kernels as they need. However, in soft real-time mode only one kernel can be created, since this kernel represents the computer where the soft real-time application is actually running. Another difference is the one mentioned about time, since for a real application the time accuracy depends

on various factors such as the hardware, the operating system, and others.

In order to be concise, only the use of JTT in EJS is explained here, although the use of JTT in Java presents big similarities to what the reader saw in the previous section.

Since the kernel and task will be executed as Java threads, the *View* or GUI of the lab can be considered a soft aperiodic task, since it will be executed only if the kernel is idle. This aperiodic task should be used to refresh the GUI (e.g. updating schedule data using the primitive `getSchedule`) and to detect any interaction of the end user (e.g. pausing the application using the primitive `pauseKernel`). When the `View` task has finished, it has to run the kernel, by executing the primitive `runKernel` without arguments.

To adjust the execution to a defined scheduling policy, the threads (the kernel and the tasks) are only executed when either of them has a Java monitor called `token`. The `sleep` and `wait` methods of Java threads, and also the method `System.nanotime` are used to control the time constraints of the schedule policy.

One limitation of this implementation is due to the fact that it is not possible to stop a thread from another thread, this option was deprecated by Java creators because it is inherently unsafe. This makes it impossible to correctly pre-empt a task that is currently executing a code segment. Hence, two options were introduced: the *concurrent* and *non-concurrent* case. In the *concurrent* case, a pre-empting thread is released and is allowed execution in *parallel* until the pre-empted thread has finished. In the *non-concurrent* case, pre-emption will not occur until the code segment has finished.

Figure 5.21 illustrates both cases. There are two tasks, `T0` with three segments and `T1` with four segments. Task `T1` has higher priority than task `T0`. The behaviour depends on the case selected:

- Non-concurrent case: Task `T1` has to wait for the end of the current segment of task `T0`. When the segment of task `T0` is finished then the first segment of task `T1` is executed. The next segment of task `T0` will be executed when the schedule policy indicates so.

- Concurrent case: Task `T1` starts execution at the release time, even though a segment of task `T0` is still being executed. The concurrence situation will finish when the task `T0` segment finishes its code (i.e. when an `EndSegment` is executed).

The next segment of task `T0` will be executed when the schedule policy decides so.

Both cases have advantages and disadvantages. The concurrent case executes the task at the release time, but the concurrence introduces an unpredictable jitter in the execution of both tasks. Moreover, pre-empting a segment can lead to unwanted side-effects. On the other hand, the non-concurrent case introduces release jitter in the execution of the tasks.

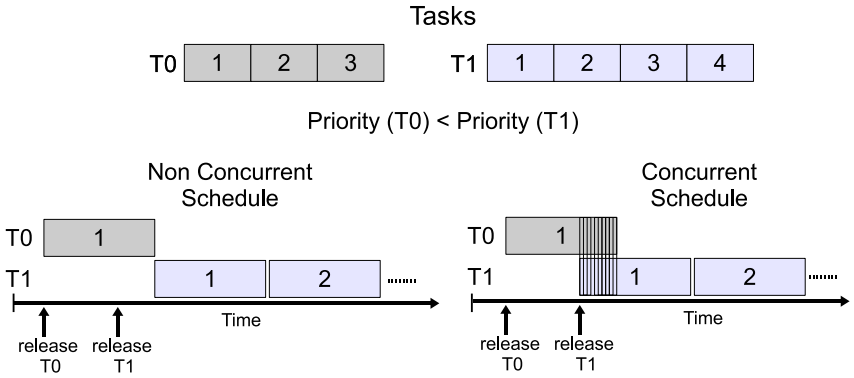


Figure 5.21: Example of concurrent and non concurrent cases. There are two tasks with three and four segments respectively. Task `T1` has a higher priority than task `T0`. In the concurrent case, task `T1` is executed even if the segment of `T0` has not yet finished, and in the non concurrent case, task `T1` has to wait for the end of the segment of task `T0`.

5.5.1 Example of a soft real-time application

An example of a real-time application is now described. This example is very simple and has no real pedagogical value. However, it shows how this mode of operation can help authors to create interactive (and even remote) real control labs. Remote labs can be described by a scheme (Figure 5.22) where the tasks require different priorities in order to provide a good performance for end users (Dormido et al. 2008).

In this example, it is supposed that just two tasks (communication and control tasks) are needed to implement the server side for a remote lab. The creation of the kernel and two tasks, `task0` and `task1`, is shown in Listing 5.14. Since in this case no ODE model is used (actually, a real plant should be used) the first parameter of `InitKernel` is null. Note that in this case the execution will use the concurrent approach.

In the creation of the tasks, the values for the periods and offsets are given in seconds. The scheduling policy is fixed priority, and the `task1` has the highest priority. Moreover, `task0` starts after one second, and one and a half seconds later, `task1` will execute its first segment.

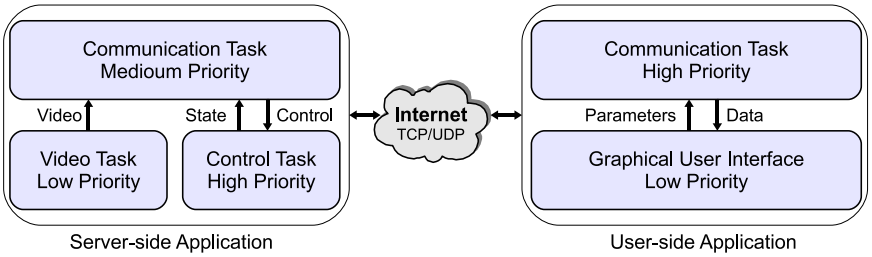


Figure 5.22: Application and tasks diagram of an interactive remote lab.

```
1  //Initialize the soft real-time system
2  RTenv.setReflectionContext(this);
3  kernel = new Kernel(false);
4  kernel.setSchedulingPolicy(Kernel.FP);
5  kernel.setSchedule(true);
6  kernel.setScheduleWindow(10);
7  kernel.setConcurrent(true);
8
9  //Create periodic Tasks
10 task0 = new Task();
11 task0.setPeriod(10);
12 task0.setPriorityValue(2)
13 task0.setOffset(1);
14 task0.addCode("code0");
15 task1 = new Task();
16 task1.setPeriod(10);
17 task1.setPriorityValue(1)
18 task1.setOffset(2.5);
19 task1.addCode("code1");
20
21 //Add tasks and kernel
22 kernel.addTask(task0);
23 kernel.addTask(task1);
24 RTenv.addKernel(kernel);
```

Listing 5.14: Creation of the kernel and tasks for the real-time application.

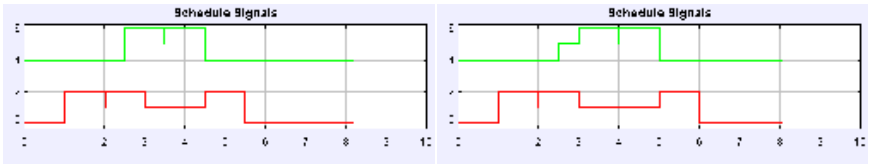
The codes for both tasks are presented in Listing 5.15. Both code functions call the method `sleepAndEndSegment()`, which generates a pause (using the Java `sleep` method) of the thread for 1000 milliseconds. After the pause, the `RTenv.endSegment` method is executed to indicate the end of the segment. Hence, `task0` has three segments of one second, and `task1` has two segments of one second.

```
1  public void code0(){
2   sleepAndEndSegment(1000);
3   sleepAndEndSegment(1000);
4   sleepAndEndSegment(1000);
5  }
6
7  public void code1(){
8   sleepAndEndSegment(1000);
9   sleepAndEndSegment(1000);
10 }
```

Listing 5.15: Code for both tasks used in the real-time application.

When the application is executed, it is possible to visualize the state of both tasks

for the first eight seconds. In Figure 5.23a, the schedule data is presented. In this case, it can be observed that the state of the `task1` (upper signal) indicates that the first segment is executed at the release time (two and half seconds). Hence, the execution of this segment is concurrent with the execution of the second segment of `task0` for half a second.



a) b)

Figure 5.23: Schedule plots for a) concurrent and b) non-concurrent cases.

If a non-concurrent case is selected for this application, the schedule signals are now different from the previous case. The results are presented in Figure 5.23b, and it can be noticed that the first segment of `task1` is delayed until the end of the second segment of `task0`, before executing.

Regarding the interactive remote lab, probably the performance of the concurrent case is better since at least the control task (the highest priority task here) will start to execute when the period is reached. This is very important, in particular, if the user introduces some disturbance in the plant.

5.6 Conclusions

The chapter describes two approaches to create virtual laboratories of embedded control systems with pedagogical purposes.

The first approach uses previous work of the thesis to add interactive human interfaces. The engineering simulations of real-time control systems are created by using the MATLAB toolbox TrueTime. The interactive user interfaces are built with the help of Easy Java Simulations as Chapter 4 shows. Although this way to build interactive simulations of real-time control systems in general produces good results, the simulation performance could be too slow in some cases. This is a consequence mainly due to the high number of events that TrueTime simulations require. In order to speed up the interactive simulations, a new feature to manipulate Simulink models from Java was added to the Simulink class. This approach ends describing in detail the implementation of some embedded control virtual laboratories.

Since TrueTime is a MATLAB toolbox, the approach described before is limited to MATLAB users, which can be an important restriction for instructors and students. For that reason, a Java library called JTT, based on TrueTime task model, has been implemented. The library is focused on providing a minimal set of features to implement educational real-time control simulations. The approach can use JTT and Easy Java Simulations together to add interactive human interfaces to the virtual laboratories developed. The approach shows some examples of use of the JTT library. Additionally, the JTT library provides some interesting features to develop soft-real time application. An example of use of this feature is also described.

Chapter 6

Experiments on Virtual Laboratories

The ultimate goal of building a simulation for a virtual laboratory is that of performing interesting experiments with the simulation. A typical definition of experiment states that *an experiment is the process of extracting data from a system by exerting it through its inputs* (Cellier 1991). This definition needs to be made more general when the experimentation system is a computer simulation. Indeed, in a computer simulation, not only are all its inputs and outputs accessible, but modern modelling tools even allow for a direct control of the model so that its behaviour can, to a certain extent, be changed in run-time. Traditionally, users of virtual laboratories are expected to perform experiments by interacting with the simulations' graphical user interface (GUI). But this frequently poses important limitations.

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

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

7. Conduct an analysis on the results thus obtained.

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

An experimentation language could be also used by teachers to run an automatic process of student's work evaluation. For instance, a control problem with different design requirements can be given to the students. They are then asked to determine the control parameters that meet the goals. Finally, teachers take the proposed control parameters and run a simulation of the control system to check if the design requirements are reached.

The next section introduces some previous experiences about performing experiments with modelling tools. The chapter then presents the main elements required by an experimentation language. These elements are implemented using the authoring tool Easy Java Simulations. Two examples are also shown using the implementation. Finally main conclusions and further work is discussed in the last section of the chapter.

6.1 Existing experimentation languages

This work defines a standard set of actions that computer simulation experiments should implement. It does so by designing an Application Programming Interface (API) or set of instructions which simulations should conform to in order to provide standard experimentation capabilities. Some modelling or simulation environments already include scripting facilities that allow users to run certain types of experiments (Brück et al. 2002, Elmqvist et al. 1998, Fritzson et al. 2002). Among them ACSL (Software 1995), EcosimPro (Internacional 2010), and Dymola (Brück et al. 2002).

ACSL stands for Advanced Continuous System Language, and it was one of the first commercially available modelling and simulation tools designed for simulating continuous systems. ACSL has been validated through more than 30 years of continuous use by

the world's most demanding simulation professionals. Nowadays its successor (acslX) and also other simulation tools have inherited its main languages features.

A simple example of the use of ACSL is described in Listings 6.1 and 6.2. The first listing shows the *implicit* program which defines a bouncing ball model. The two simple differential equations which involve the velocity and height are computed using the command `INTEG`.

```
1  DERIVATIVE !bouncing ball
2    CONSTANT   v0=0.0    ,   h0=10.0  !initial conditions
3    CINTERVAL  cint=0.1               !communication interval
4    !equations
5    velocity = INTEG(gravity,v0)
6    height   = INTEG(velocity,h0)
7    !stop condition
8    TERMT(t.ge.20, 'Time Limit')
9  END !of derivative
```

Listing 6.1: Bouncing ball model.

The model is simulated by using the commands of Listing 6.2. Here a procedure, called `simulateball`, is defined to simulate and plot the height and velocity of the bouncing ball model (defined in Listing 6.1). The procedure is then invoked after setting the initial height for 10.0 and 20.0 meters as the listing shows.

```
1  SET TITLE = 'Bouncing ball example'
2  SPARE
3  OUTPUT t,velocity,height/NCIOUT=5 !List to be printed during run
4  PREPARE t,velocity,height         !List to be saved for later use
5  PROCEDURE simulateball
6  START
7  PLOT velocity,height
8  END ! of procedure simulateball
9  SET h0=10.0; simulateball         !Plot a first simulation
10 SET h0=20.0; simulateball         !Plot a second simulation
11 ....
12 SPARE
13 QUIT
```

Listing 6.2: Bouncing ball run time commands.

The script of Listing 6.2 shows how to manipulate programatically the simulation in ACSL. Dymola also support a script facility that makes it possible to load model libraries, set parameters, set start values, simulate and plot variables by executing scripts. The script facility is useful when running a series of simulations such as a parameter study. Listing 6.3 shows an example of the Dymola's scripts, here the bouncing ball model is run ten times for different heights. Dymola's scripts facility allows very little flexibility, since there is no possibility to manipulate the simulation while this is being executed.

```
...
//Open the model
openModel("bounce.mo");

//Compile the model
translateModel("bounce.mo");

//Run the experiments
for i in 1:10 loop
 bounce.height=i*10;
 bounce.velocity=0;
 simulateModel("bounce", startTime=0, stopTime=10,resultFile="out"+String(i));
end for;

//Close the model
closeModel();

//Process the result files
...
```

Listing 6.3: A script to perform an experiment with Dymola.

The work described in this chapter has been inspired by these previous experiences but has also added its own requirements to create a universal, full-fledged specification that provides more general and flexible features.

6.2 Defining an experimentation language

In order to test the viability of the language, it has been implemented using the authoring tool Easy Java Simulations (EJS). EJS is used due to the facilities that this software provides to create interactive simulations in Java, described in Chapter 4. Furthermore, the tool can incorporate existing Java packages, such as JIMC or JTT, to create the simulation. The goal is that these interactive simulations implement the experimentation language. Further details about the implementation of the experimentation language on EJS can be found in (Esquembre et al. 2007).

6.2.1 Elements of an experimentation language

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

- Elements to run one or more instances of a simulation.
- Elements to access variables and routines.
- Elements to specify algorithms.

- Elements to control the execution of the simulation.
- Elements for user input.
- Elements to allow for comparison of results.

These categories are discussed now in more detail.

Elements to run one or more instances of a simulation

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

Users should also be able to specify whether they want the running simulations to be executed either *synchronously* or *asynchronously*.

Synchronized simulations advance (step) through their evolution cycle at the same pace. In particular, if the simulations use the same increment of time for each step, their internal time will remain synchronized. The synchronous case can be useful to highlight differences between the simulations' outputs that run at the same time but with distinct initial conditions.

Asynchronized simulations are stepped at different moments, advancing each time when the experiment requires it. This case could be interesting to start or pause a simulation instance in response to an experiment condition or to some user inputs.

Elements to access variables and routines

Users need to be able to read and to set the value of the variables of the model of a simulation at any time. This can only be restricted if the simulation designer has declared some of the model variables as non-accessible (private). The same principle applies to routines or functions (methods) that the simulation defines. Users should be able to easily obtain information about available variables and methods. Notice that, at first glance, in principle, the declaration of some methods or variables as private could be considered a restriction of the full controllability and observability required by the experimentation language. However, the encapsulation (information hiding) has to be considered a good principle of the simulation design, since it can help protect the

correctness of the model behaviour by adding a suitable interface to manipulate the simulation.

Elements to specify algorithms

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

Elements to control the execution of the simulation

Users may want to control the simulation execution. This includes not only standard play and pause instructions that start/stop the simulation, but also instructions to run the simulation until a given condition is met, such as (in human language):

```
run simulation until the tank's level is higher than ten centimeters.
```

The experimentation environment should then be able to hand over the control of the computer resources to the running simulation and wait until the simulation meets the given criteria and pauses, giving then control back to the experiment.

Another feature required is the possibility of planning events in the future, such as:

```
run simulation increasing the set point by 50% when time is ten seconds.
```

This can be especially useful for instructors that want to generate demonstrations (demos) of the use of the simulation in order to describe some specific functionality of the virtual lab to the students.

Elements for user input

On occasion, partial results of the experiment may require user input. Functions in this category should allow displaying messages or asking users to enter one or more numeric values, choose a given option out of several offered, or confirm an action. Besides, if it is necessary, the experiment can provide the possibility that end users interact with the GUI of the simulations at any time.

Elements to allow for comparison of results

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

6.3 Implementation

Easy Java Simulations has been chosen for the implementation because it offers several appropriate characteristics as commented in Chapter 4. EJS falls into the category of code generators, which makes it possible to use all the constructions provided by a standard programming language. The fact that EJS is based on Java has also been crucial in this work because it helps manage several instances of a simulation, or address compound objects (such as graphs) in them, in an object-oriented way. Finally, users of EJS can easily inspect, understand, and, if necessary even modify, other people's simulations, which greatly increases their observability and controllability.

The possibility of defining experiments for existing simulations has been added to EJS by loading the XML file that describes the simulation (which may have been created by another person) and adding pages defining experiments for it. When the simulation is re-generated, it adds to its standard menu an entry for each of the experiments thus defined. Users simply select the experiment as one pop-up menu option. When running the simulation as an applet, the experiments can also be accessed using hyper links embedded in the HTML page that contains the simulation. This feature provides a way to include, in a very natural way, the execution of experiments on the simulation in curricular material developed in HTML web pages.

To implement the experimentation language, new built-in methods have been added to Easy Java Simulations that provide the necessary functionality. Figure 6.1 shows the new panel of EJS, called Experiments, where the experiments for the simulation can be described.

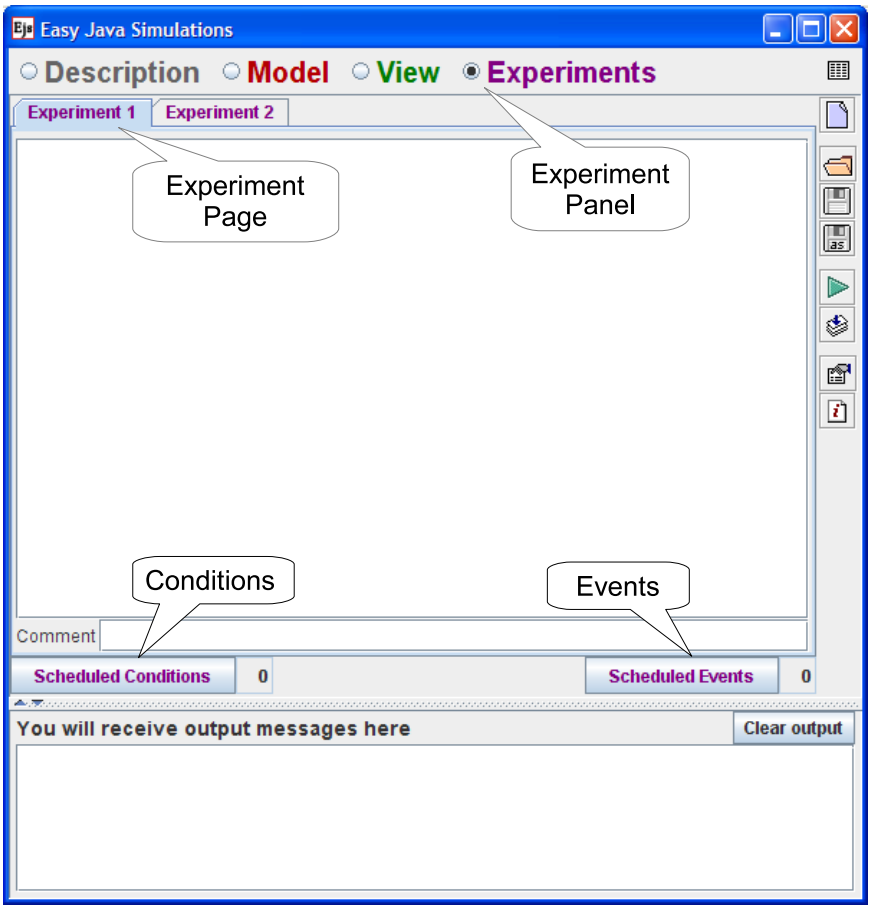


Figure 6.1: The Experiment panel in EJS.

EJS uses the code of an experiment, described in the panel, to generate an inner class that extends the abstract class `Experiment`, shown in Listing 6.4. The generated class overrides the method `run` of `Experiment` with the experiment's code.

```
1  ...
2  public abstract class Experiment implements Runnable {
3    ...
4    public Experiment (String _name, String _description) {
5      name = _name;
6      description = _description;
7    }
8
9    public void _runExperiment(){
10     if (_thread!=null) _stopExperiment();
11     _thread = new Thread(this);
12     _thread.setPriority(Thread.NORM_PRIORITY);
13     _shouldStop = false;
14     _thread.start();
15   }
16
17   public void _abortExperiment() {
18     ...
19   }
20
21   abstract public void run ();
22   ...
23 } // End of class
```

Listing 6.4: The Java abstract class for experiments.

The next section describes how the implementation has been done for each category of the experimentation language.

6.3.1 Elements to run one or more instances of a simulation

The Application Programming Interface provides two instructions to create a running instance of a simulation:

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

These are instance methods of a predefined object called `_simulation`, which points to the simulation itself. The first method creates and runs a copy of the simulation from which the experiment is initiated, because all simulations created with EJS extend the abstract class `Simulation`. The second method creates a copy of the simulation with the given class name. Every Java simulation is an object of a given class and several classes can be packaged together in compressed archives called JAR files. Users can instantiate any simulation which is in the same JAR file as the original simulation or in any other JAR file included in the simulation's class path. EJS simulations can add JAR files to their class path using the field `Imports` in the Information Panel of EJS as was described in Chapter 4.

Listing 6.5 shows the implementation in the class `Simulation` of both methods to run the simulation.

```
1  public abstract class Simulation implements Runnable, ActionListener {
2
3   public Model runSimulation(){
4     return runSimulation(null);
5   }
6   public Model runSimulation(String classname){
7    try {
8      Class theClass;
9      if (classname==null) theClass = getModel().getClass();
10     else theClass = Class.forName(classname);
11     Model simModel = (Model) theClass.newInstance();
12     Simulation top = getTopMaster();
13     simModel.getSimulation().master = top;
14     simModel.getSimulation().isPlaying = isPlaying;
15     simModel.getSimulation().update();
16     top.slaveList.add(simModel);
17     return simModel;
18   }
19   catch (Exception exc) {
20     exc.printStackTrace();
21     return null;
22   }
23  }
24 ...
25 }
```

Listing 6.5: Instance methods defined in the abstract Java class Simulation.

Simulations created using either of these two methods appear automatically on the computer screen and are by default synchronized with (i.e., they are subordinates of) the original one. Subordinates of a simulation can be freed (made to run asynchronously) using the `_simulation` instance method:

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

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

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

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

6.3.2 Methods to access variables and routines

Experiments are created and run as part of the model of a simulation. This gives them direct access to the model's variables and methods. Both versions of the `runSimulation` method described above return an object of the corresponding model class, which is an implementation of the generic Java interface `org.opensourcephysics.ejs.Model`, included by default in every EJS simulation's JAR file. Users need to typecast this object into a local variable of the correct type in order to access the model's public variables and methods. The standard object-oriented *dot* mechanism of Java can then be used to address any variable or method in the simulation model. For example, suppose that an experiment from a `MySimModel` class simulation has been run, having a variable called x and a method called action. The experiment can then use constructions of the form shown in Listing 6.6.

```
1  // Create a subordinate instance of this simulation
2  MySimModel sub = (MySimModel) _simulation.runSimulation();
3  x = 1.0;        // Sets the x variable of this simulation
4  action();       // Invokes the action method of this simulation
5  sub.x = 0.0;    // Sets the x variable of the subordinate
6  sub.action();   // Invokes the subordinates action method
7  _play();        // Plays both simulations synchronously
```

Listing 6.6: Accessing model's members.

6.3.3 Elements to specify algorithms

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

6.3.4 Elements to control the execution of the simulation

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

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

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

```
void _playAndWait();
```

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

Listing 6.7 shows the implementation of the methods `_playAndWait` and `_pause`. Both methods call the respective methods to play and pause the simulation. However, they also executes the methods `_controlForSimulation` and `_controlForExperiment` respectively from an instance of the class `SimulationExperiment` shown in Listing 6.8. As their names indicate, the last two methods are used simply to synchronize(by calling `wait` and `notify` functions) the execution of the two threads, the simulation and the experiment.

```
1  public void _playAndWait(){
2   _simulation.play();
3   _cSE._controlForSimulation();
4  }
5  public void _pause() {
6   _simulation.pause();
7   _cSE._controlForExperiment();
8  }
```

Listing 6.7: Methods to control the simulation.

```
1  private class SimulationExperiment {
2    public synchronized void _controlForSimulation() {
3      try { wait(); }
4      catch (Exception _exc) {}
5    }
6    public synchronized void _controlForExperiment(){
7      notify();
8    }
9  }
```

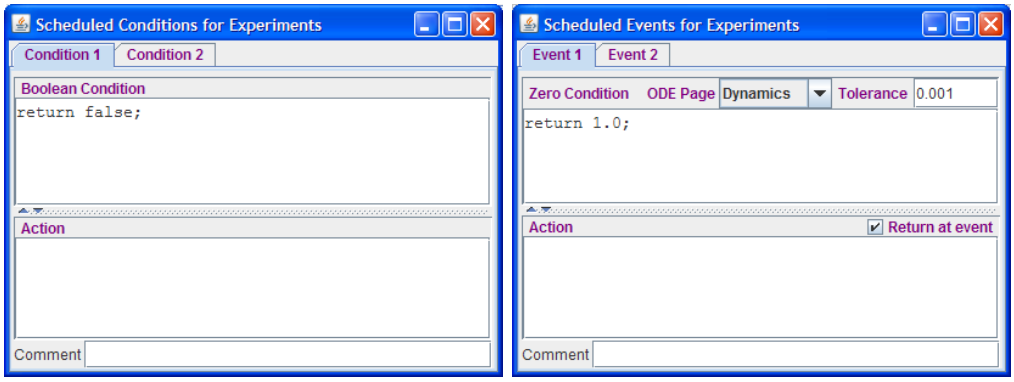
Listing 6.8: Inner class SimulationExperiment.

Once the simulation is playing, it can be paused by either user interaction, an invocation of the `_pause` method included in the original simulation, or by using one of the following new methods:

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

These two methods introduce the possibility of executing code whenever a given condition is satisfied. This code can be used to simply pause the simulation or to execute other more complex actions. The parameter of both instructions refers to an instance of one of the new constructions called *scheduled condition* and *scheduled event*, respectively, which can be defined using a special editor provided by EJS.

Both constructions consist of two methods each. The first method determines whether a given condition is satisfied by the model state. The second method allows a user-defined action that will be invoked when this condition is met. Figure 6.2 shows the editors to add the functionality.



a) b)

Figure 6.2: a) Editor for scheduled conditions, b) Editor for scheduled events.

There are some differences between both constructions. Scheduled conditions (see Figure 6.2a) are determined by a method returning a boolean value, which is tested after every simulation step. If the method returns a true value, the corresponding action is

executed. Scheduled events (see Figure 6.2b) are associated to any of the systems of ordinary differential equations (ODEs) defined by the model as part of its evolution algorithm, and are triggered by the change in sign of a positive function of the variables involved in that system of ODEs. When the function returns a negative value, the simulation detects the event, goes back to the exact instant in time when the function crossed zero, and applies the event action at that instant. In this sense, scheduling an event is similar to adding new events to the original system of ODEs in runtime. In contrast with normal events, though, scheduled events (and scheduled conditions, as well) disable themselves automatically once they take place.

6.3.5 Elements for user input

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

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

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

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

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

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

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

6.3.6 Elements to allow for comparison of results.

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

```
void clear();
```

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

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

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

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

6.4 Examples of experiments

In this section two examples of experiments created for a simulation of the PI (proportional integrator) control of the level of a tank. The simulation's typical behaviour for default `Kp` and `Ti` values of the PI controller is shown in Figure 6.3.

$$\frac{d\,level}{d\,t} = \frac{-a}{A}\sqrt{2 \cdot g \cdot max(level, 0)} + \frac{K_{flow}}{A} \cdot u \tag{6.1}$$

The dynamics of this single tank simulation, given by Equation (6.1), is determined by the `Dynamics` page of ordinary differential equation shown in Figure 6.4.

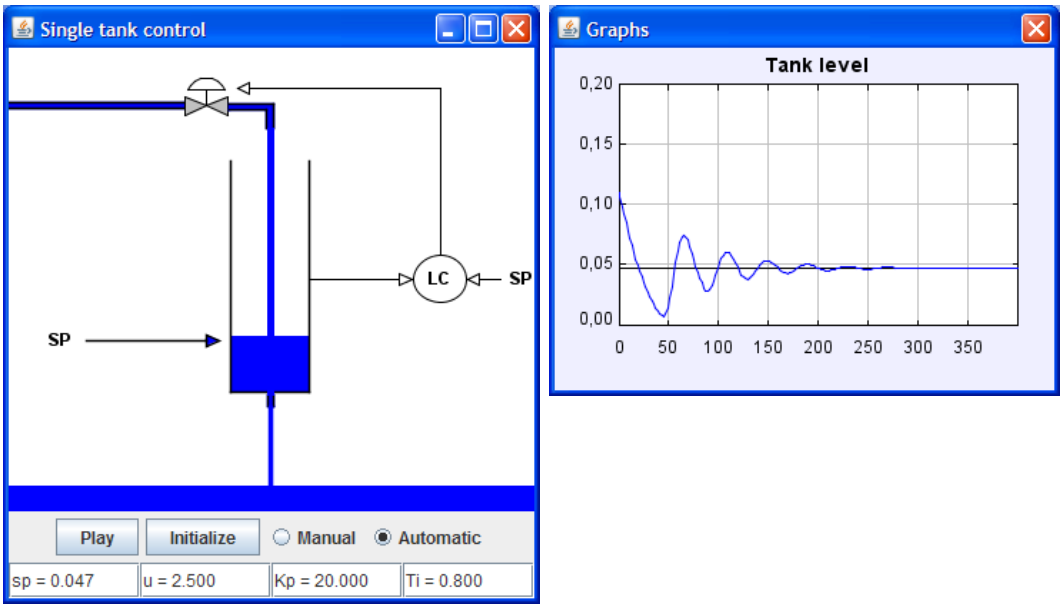


Figure 6.3: Typical response of the single tank simulation.

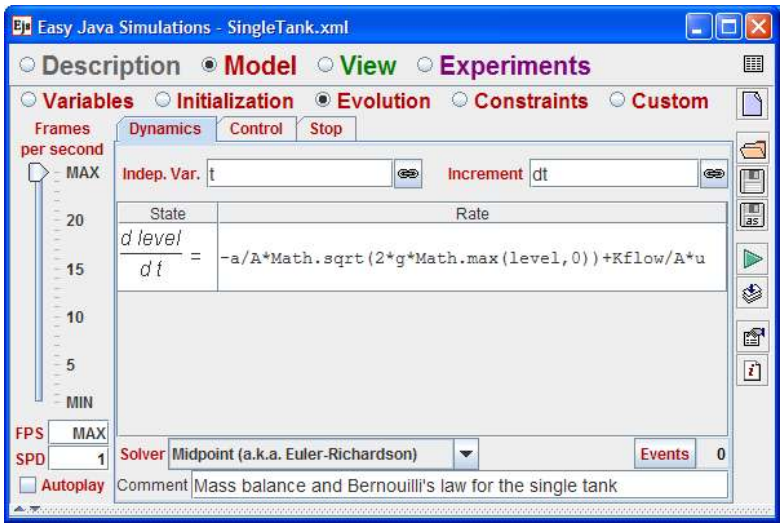


Figure 6.4: Dynamic equations of the single tank system.

The control signal `u` is computed in the second page of the evolution of the model using the code described in Listing 6.9.

```
1  if (automaticMode) {
2    // P + I action
3    u = Kp*(setPoint - level) + integral;
4    if (u<0) u = 0;
5    // Update integral action
6    integral = integral + Kp*dt/Ti * (setPoint - level);
7  }
```

Listing 6.9: Computing the control action.

which implements a digital PI controller.

6.4.1 Experiment I. Executing a scheduled event

Here a very simple experiment is described; it consists in doubling the set point when the time equals 200 seconds is described. For this, a scheduled event in the dedicated panel of EJS is first defined, as shown in Figure 6.5.

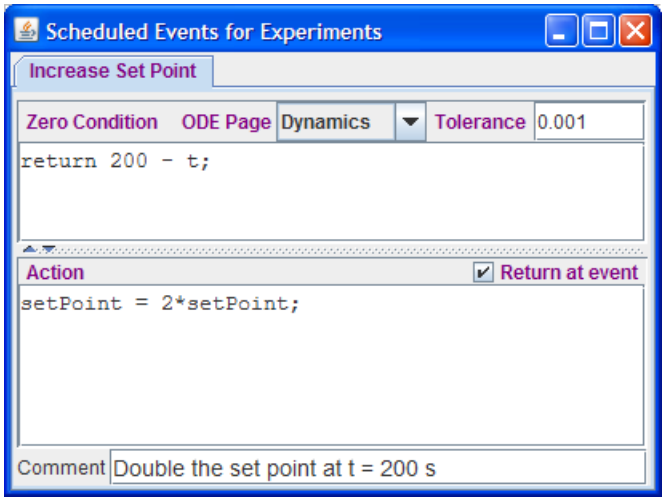


Figure 6.5: An scheduled event to change the set point at 200 seconds.

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

Then, a new page with the code displayed in Figure 6.6 is created in the panel for experiments in EJS' interface.

Once the simulation is run the pop-up menu of the main drawing panel includes an entry for the experiment. See Figure 6.7 (Experiment II defined in the next subsection is also displayed).

Selecting this experiment in the menu produces the results of Figure 6.8.

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

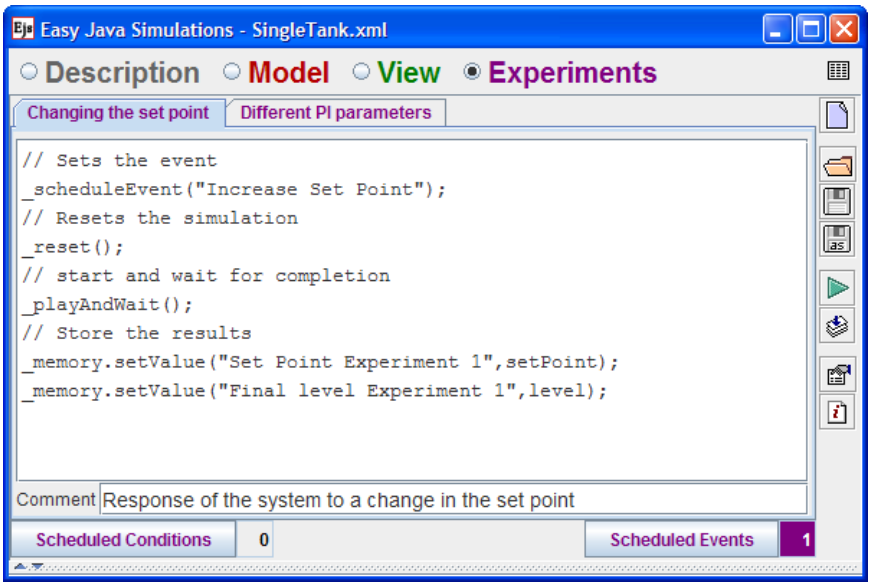


Figure 6.6: Definition of experiment I in EJS.

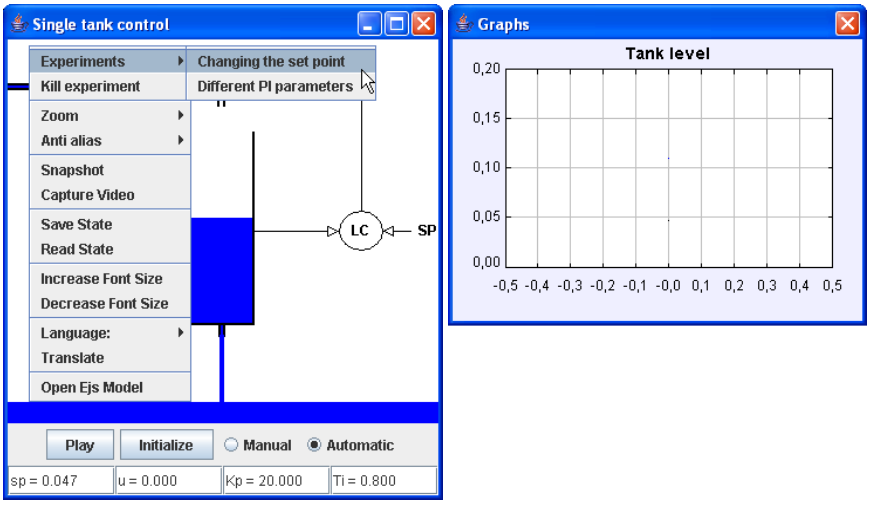


Figure 6.7: Running experiment I from the simulation interface.

6.4.2 Experiment II. Comparing graphic outputs

In this example, the responses of the PI control with different `Kp` and `Ti` parameters are compared. A simplistic solution would be to run the simulation manually twice, once for each set of parameters, take snapshots of the evolution graphs, and then compare them looking at each graph side by side. A better procedure, though, is to conduct an experiment that automatically creates a second copy of the simulation, changes its parameters, and then runs both simulations synchronously, displaying the graph of their responses in the same plot. The experiment code is shown in Listing 6.10.

```
1  _reset(); // Resets the simulation
2  // Creates a subordinate simulation
3  SingleTank subordinate = (SingleTank) _simulation.runSimulation();
```

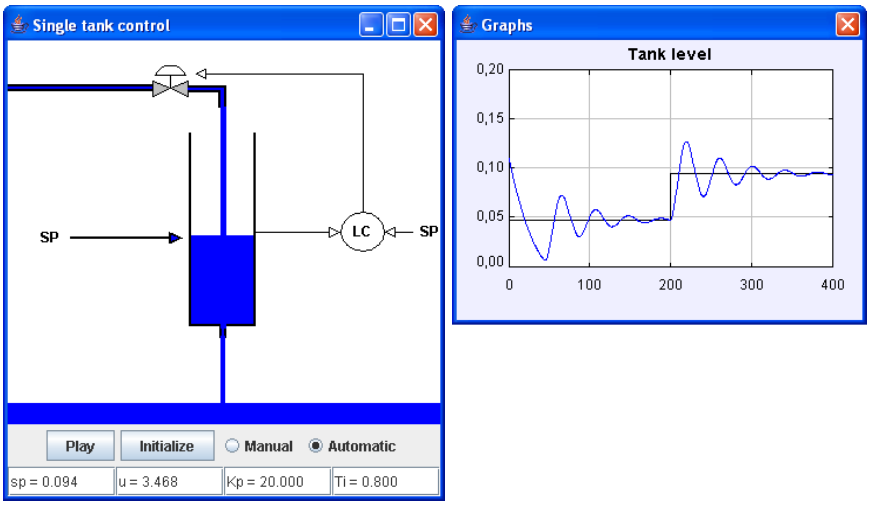


Figure 6.8: Output of experiment I.

```
4  subordinate.Kp = 30;  // Sets the subordinates Kp
5  subordinate.Ti = 1.0;  // Sets the subordinates Ti
6  java.awt.Color color = java.awt.Color.RED; // Chooses a color
7  // Changes the color of the subordinates level trace
8  subordinate._view.levelTrace.getStyle().setEdgeColor(color);
9  // Reparents the subordinate's level trace into the plotting panel
10 subordinate._view.reparentDrawable("levelTrace",
11       _view.getElement("plottingPanel"));
12 subordinate._view.dispose(); // Hides the subordinate's view
13 _play(); // plays both simulations
```

Listing 6.10: Experiment II.

The output of this experiment is shown in Figure 6.9.

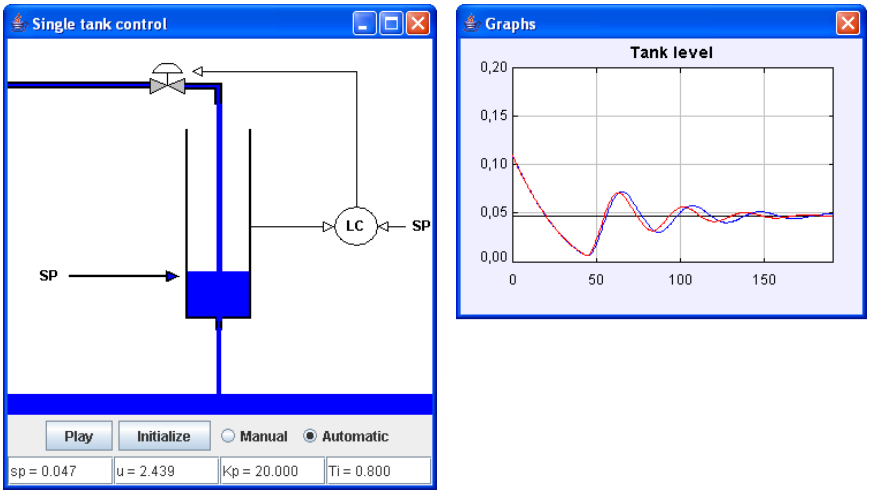


Figure 6.9: Output of experiment II.

6.4.3 Advanced Experiments

Instructors can develop advanced experiments to teach control engineering by using the experimentation environment and some Java programming knowledge.

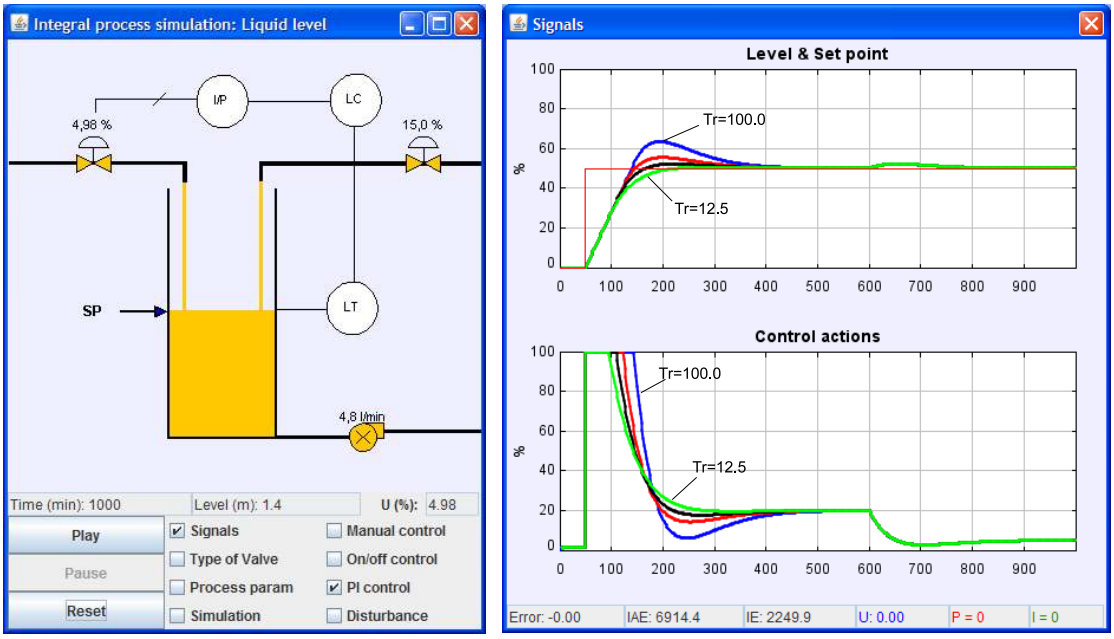


Figure 6.10: System responses with anti-windup method. The results shows the influence of `Tr` on the output.

Figure 6.10 depicts the result after execute the experiment described in Listing 6.11. This experiment uses a simulation of a single tank (modelled as an integrator) controlled by a PI controller to demonstrate the influence of the reset time `Tr` parameter on controllers with anti-windup. The same simulation is run four times but with different values of `Tr`. System responses with bigger values of reset time shows worse performance. The anti-windup techniques are required to avoid the saturation of the integral term of a PID controller due to the limitations of actuators (Åström & Hägglund 2005).

```
1  //Prepare the experiment
2  _alert ("PlayField","Experiment 1","Varying Tr in a PI controller");
3  _scheduleEvent("setpoint 2");
4  _scheduleEvent("disturbance 2");
5  ...
6  _reset();
7
8  //Run the experiment
9  Tr=Ti=100.0;
10 antiwindup=true;
11 for (int i=4;i>0;i--){
12   if (i==4) {
13       _view.Level.getStyle().setEdgeColor(java.awt.Color.BLUE);
14       _view.Trace_U.getStyle().setEdgeColor(java.awt.Color.BLUE);
15   }else if (i==3) {
16       _view.Level.getStyle().setEdgeColor(java.awt.Color.RED);
17       _view.Trace_U.getStyle().setEdgeColor(java.awt.Color.RED);
18   }else if (i==2) {
19       _view.Level.getStyle().setEdgeColor(java.awt.Color.BLACK);
20       _view.Trace_U.getStyle().setEdgeColor(java.awt.Color.BLACK);
21   }else {
22       _view.Level.getStyle().setEdgeColor(java.awt.Color.GREEN);
23       _view.Trace_U.getStyle().setEdgeColor(java.awt.Color.GREEN);
24   }
25   _playAndWait(); // plays the simulation
26   _initialize();
```

```
27    connect=false;
28    t=u=xc=setPoint=setPoint_meters=level=I=P=0;
29    d=15;evaluateDisturbance(); checkDisturbance();
30    Tr=Tr/2.0;
31  }
32
33  //Finish the experiment
34  _alert ("PlayField","Experiment 1","End of experiment");
```

Listing 6.11: Experiment to evaluate the influence of reset time of the anti-windup method.

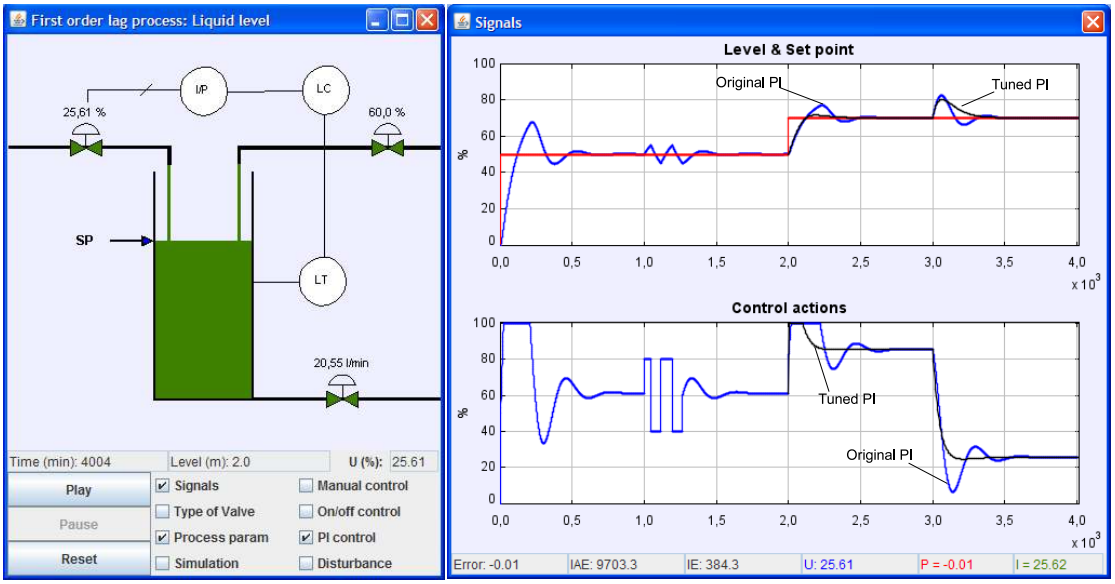


Figure 6.11: Auto-tuning of PI controller.

Other advanced experiment, described in Listing 6.12, is depicted on Figure 6.11. In this case the experiment uses a simulation of a single tank (but now modelled as a first order system) controlled by a PI controller to demonstrate the relay tuning method (Åström & Hägglund 2005). The simulation starts with the tuning phase at time = 1000. The amplitude and period of the system response is used to tune the PI controller. After time = 2000, both controllers the original and the computed by the tuning method are used to control the system.

```
1   //Prepare the experiment
2   _alert ("PlayField","Experiment 8","PI parameters"+" Kp:"+K+" Ti:"+Ti);
3   _scheduleEvent("setpoint 2");
4   _scheduleEvent("manual");
5   _scheduleEvent("relayOff");
6   _scheduleEvent("relayOn");
7   ...
8   _reset();
9
10  //Create a subordinate
11  subordinate = (SingleTank) _simulation.runSimulation();
12  ...
13
14  //Compute PI parameters
15  experimentOn=true;
16  _playAndWait();
17  cycleAT=t-cycleAT;
```

```
kcAT=4*1/(Math.PI*Math.sqrt(0.2*0.2));
experimentOn=false;

//Test new PI parameters
onoffMode = false; PIDMode = true; manualMode = false;
setPoint=50;
setPoint_meters=FS_out * setPoint / 100.0;
subordinate.K=0.5*kcAT;
subordinate.Ti=2*cycleAT/Math.PI;
subordinate.evaluatePID();
...
_playAndWait(); // plays the simulation

//Finish the experiment
_alert ("PlayField","Experiment 8","End of experiment");
```

Listing 6.12: Experiment to demonstrate the auto-tuning method of a PID.

The advanced experiment described here, just show the possibilities provided by the experimentation environment implemented in Easy Java Simulations. Other advanced experiments can be obtained for instance by combined the experimentation environment with interoperate approach or with the use of JTT library.

6.5 Conclusions

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

More generally, the API can be the basis for the definition of a standard experimentation language to which other modelling and simulations tools could adhere. This is the goal in the coming future. The current, experimental version of EJS that supports the features described in this paper can be downloaded from:

`http://www.um.es/fem/publications/2007/Ejs070507.zip`

Chapter 7

Conclusions and Future Works

Information and communication technologies have a widespread impact on modern life and education is no exception. These technologies have enabled many advances in an increasing number of fields, such as the Internet, web services, video conferences, web-based courses, interactive graphics, and others.

Control education has also been affected positively by these technologies. Virtual and remote laboratories are increasingly being used to enhance the way in which students interact with simulated or real resources. This interaction offers new learning elements without the typical time, spatial, or pace constraints of traditional laboratories.

Although a lot of advances in these computer-based laboratories have been done for the scientific community. There is room for improvement.

For instance, many simulations do not allow user interaction while the simulation is running, which forces students to wait until the end of the simulation to be able to experiment the system with different parameters. Other simulations provide simple plots of the signals of the model to show the behaviour of the system, which are not explanatory enough for students. However, the advanced graphical power of modern computers can provide more flexible and human-readable learning material.

These interesting features can be added, as a human interface layer, to engineering simulations in order to ease the learning process and to reduce the time required to gain insight into fundamental engineering concepts.

However, the creation of computer-based laboratories with advanced human interfaces is not an easy task. Many engineering software provide good libraries to build engineering simulations, but they normally lack tools to add advanced human interfaces. Thus, the creation of these laboratories can demand great effort of instructors,

especially those who are not experts at computer programming.

On the other hand, once instructors finally create an appropriated graphical user interface for their simulation, they might get frustrated by the fact that such interfaces are normally incompatible between engineering software.

This thesis focused on these problems, and its main contributions consist of new tools for instructors to create virtual and remote laboratories in an easier way.

In order to generate a standardized way to add human interfaces to existing models, Chapter 2 introduced a novel design approach for building interactive engineering simulations. The interoperate approach splits the creation of computer-based laboratories into two separated activities.

First, the instructor develops the engineering simulation using a standard engineering software. Then, the instructor uses a specialized programming language, such as Java, to create the interactive human interface. Both components are then integrated using a generic communication protocol to manipulate the engineering simulation from the human interface.

The thesis describes the standardized communication protocol required to support the interoperate approach. The protocol has both a high and a low-level specification to manipulate the external application (the engineering simulation) from the client application (the human interface). The high-level protocol offers authors the control of the external application at a simple, high level of abstraction, hiding a number of details, but still providing an effective link between the engineering model and the human interface of the simulation. The low-level protocol gives authors total control on the external simulation, providing an enhanced link between both simulations, bringing a richer level of interaction and visualization.

The high-level protocol is all that most authors will need to implement their interaction requirements, it being therefore the recommended entry level for authors who are not expert programmers or do not need a very detailed control of the communication between client and external applications. The low-level protocol is the preferred choice for authors that need full control of the original simulation and the communication mechanism. However, the use of the low-level protocol requires some more programming efforts than that of the high-level protocol.

The standardized communication protocol also allows the control of engineering sim-

ulations over networks. Thus, the interoperate approach can be used to create interactive local and remote laboratories in a uniform way. In fact, the student using a virtual laboratory designed with the interoperate approach, will not observe any difference between a local and a remote version of the same computer-based laboratory, except for network delays. Such delays prompted the development of two types of remote links for the high-level protocol: synchronous and asynchronous remote link.

In Chapter 3 the implementation of the interoperate approach for various well-known engineering software is described. This implementation follows a similar scheme for all cases since most engineering software provides an interface to be called from external languages such as C or Java. Using this interface, the engineering software can be controlled from a Java class.

The Java classes implemented in this thesis allow authors to manipulate applications such as MATLAB and Scilab according to the standardized protocol of communication designed previously. Since the control of engineering simulations is now standardized, human interfaces developed for a MATLAB simulation can be reused without any modification for a Scilab simulation.

Some of the Java classes implemented conform the free **JIMC** Java package. Instructors can use this library to create virtual and remote laboratories using the MATLAB/Simulink software. The **JIM** server can also be used to support a remote interaction with MATLAB/Simulink simulations.

Once the manipulation of the engineering software is established according to the interoperate approach, instructors can use Java packages, such as `Swing` and `AWT`, to create the human interface of the interactive laboratory. The construction of this graphical user interface typically demands a lot of effort from the implementation standpoint, especially for non-programming instructors. For this reason, instructors can use some specialized Java authoring tools, such as Easy Java Simulations (EJS), to build the interactive user interface in a much easier way.

The creation of user interfaces and the use of the interoperate approach from EJS is described in Chapter 4. This chapter is divided into two parts. The first part describes the use a the Java package, such as JIMC, to communicate with engineering software from EJS. The second part presents a new version of EJS which integrates all the Java classes described in Chapter 3. Instructors can use this version of EJS to create, even

more easily than before, interactive laboratories with MATLAB, Simulink, Scilab, or Sysquake simulations. The two parts of the chapter provide simple examples of the use of EJS under the interoperate approach. The chapter ends with a real-world example of a networked control laboratory used in an introductory course of engineering control at Ghent University in Belgium.

After the description of how to add human interfaces to any engineering simulation, the thesis considers in detail the simulation of embedded control systems. These systems are the subject of recent interest because, contrary to the traditional design, a novel approach of analysis considers the real-time and control theory together. This new perspective provides a perfect case study for the interoperate approach thanks to TrueTime MATLAB/Simulink-based toolbox.

Chapter 5 discusses the creation of virtual laboratories of embedded control systems. The chapter is also divided into two parts. The first part describes the creation of interactive simulations using the interoperate approach. Authors can create engineering simulations of real-time control systems using TrueTime functionalities, and then move to EJS to add the interactive human interface, as Chapter 4 described. The second part of Chapter 5 shows the implementation of the **JTT** Java package based in TrueTime features. The main reason for this library is to provide instructors with a free solution to create interactive simulations of real-time control systems for educational purposes. Both parts of Chapter 5 show several examples of the use of the tools described.

The thesis finally considers a third topic, i.e. the creation of experiments with virtual laboratories. The experiments can be used, for instance, to optimize model parameters by running many instances of the simulation. This problem has been considered since the first computer-based simulations. However, the creation of experiments has evolved very little since then. The thesis proposes to exploit the capabilities of modern languages to perform educational experiments with simulations.

Chapter 6 describes a set of elements needed to perform modern experiments. An implementation of a basic experimentation environment is also provided. This environment has been implemented in a version of Easy Java Simulations. Instructors can create experiments in a new section of EJS called `Experiments`.

Some examples of use are presented to highlight main features of the proposed experimentation environment. From the educational standpoint, the experiments can be

used to compare at run time the simulation outputs under different scenarios.

Future works

Although this work has provided many important results, there is still room for improvement. Future work is possible in the three topics treated in the thesis.

The interoperate approach has proved its validity for four different engineering software. The scheme used in Chapter 3 can be easily followed to implement the communication protocol for other external applications such as Octave, Maple, and Dymola. In fact, much of the engineering software presents modern external interfaces that can provide access from Java programs. It could also be interesting to implement a dedicated communication protocol for the Scicos toolbox of Scilab. Scicos is a block diagram simulator close to Simulink. Thus, based on the experience with Simulink simulations, a special link could also be implemented for Scicos.

Although the implementation of the communication protocol has been done using the Java language, it could be interesting to test the validity of the interoperate approach using another general purpose language such as C, C++, or C#.

With regards to the remote operation of the communication protocol, in this thesis only MATLAB and Simulink have been treated to support remote links. Other engineering software, such as Scilab and Sysquake, could also support an implementation of the remote operation, following the scheme described in this work.

The networked control laboratory described in Chapter 4 presents a real-world example of a remote laboratory. Even though this example shows that this kind of applications can be created by using the communication protocol, there is still a lot of work to be done. For instance, using a differentiated network protocol (TCP/IP or UDP) to treat the incoming and outcoming data of the remote laboratory.

In low-bandwidth or congested networks, the use of the TCP/IP network protocol can add unnecessary delays to the data traffic from the real plant. The delays are mainly due to the congestion and flow control of the TCP/IP protocol. By contrast, the UDP protocol can provide a faster data exchange between client and server, but at the expense of losing the error-free mechanism that the flow control offers.

On the one hand, the incoming data flows from the remote server to the client application transporting real data of the plant. This data is mainly used by the user

interface to monitor the state of the real plant. Thus, instead of using TCP/IP to transport incoming data, the communication protocol could use the UDP network protocol to improve the performance of the remote laboratory.

On the other hand, the outcoming data flows from the client application, the user interface, to the remote server transporting mainly control data. This data can be used either to modify the parameters of the controller or to send the control action to the real plant. Thus, the selection of TCP/IP or UDP should be done considering whether or not the control of the real plant depends on error-free outgoing data.

The interoperate approach could be extended also to other platforms such as mobile devices, in order to benefit from the computational power of mobile phones or from the new user experience provided by *touch-screen* computers.

With regards to the simulation of embedded control systems, the performance of the two approaches presented in this thesis can be optimized.

The TrueTime-EJS approach still requires a high number of zero-crossing functions to simulate real-time control systems in Simulink. Thus, new methods to reduce the event evaluations or to speed up the simulation could be added to the communication protocol that manipulates Simulink models from Java programs.

Besides, the communication protocol could consider a dedicated link with the C++ implementation of TrueTime in order to improve TrueTime simulations. The TrueTime implementation on Scicos can be also analyzed in order to provide an open source approach to the creation of interactive simulation of real-time control systems.

The JTT-EJS approach can also be extended to support wired and wireless communication networks in the virtual laboratories created with it. Other functionalities, such as the battery block of TrueTime could be also added to the JTT Java package.

Some examples of the use of the functionality to simulate soft real-time systems could be also considered in the future. The use of EJS, JTT, and the collection of open-source drivers of the Comedi project, can be used to offer a complete open-source platform to develop remote laboratories.

Both approaches, TrueTime and JTT, can also be used to generate new virtual laboratories in order to get a larger supply of simulations of real-time control systems. Moreover, there laboratories could be integrated into a course of engineering control or real-time systems.

Future work on experiments on virtual laboratories will mainly involve the development of new applications.

The experimentation environment could also be extended to create experiments on remote laboratories. This feature is interesting, educationally, to show students the effects of an incorrect selection of control parameters when controlling a real plant.

Experiments on remote laboratories can also be used to create generalized algorithms to identify the model of a real plant.

The combination of the experimentation environment with the interoperate approach and the real-time simulation can also be an interesting research topic in order to extend the application field of the experiments.

Finally, in a more general context, the described API can be the basis for the definition of a standard experimentation language to which other modelling and simulations tools could adhere.

Bibliography

Andersson, M., Henriksson, D., Cervin, A. & Årzén, K. (2005), Simulation of wireless networked control systems, *in* 'Proceedings of the 44th IEEE Conference on Decision and Control and European Control Conference ECC'.

Åström, K. & Hägglund, T. (2005), *Advanced PID Control*, Research Triangle Park, NC: ISAThe Instrumentation, Systems, and Automation Society.

Åström, K. & Witternmark, B. (1997), *Computer-controlled systems*, 3rd edn, Prentice Hall.

Balestrino, A., Caiti, A. & Crisostomi, E. (2009), 'From remote experiments to web-based learning objects: An advanced telelaboratory for robotics and control systems', *Industrial Electronics, IEEE Transactions on* **56**(12), 4817 –4825.

Brück, D., Elmqvist, H., Mattsson, S. & Olsson, H. (2002), Dymola for multi–engineering modeling and simulation, *in* 'Proceedings of the 2nd International Modelica Conference'.

Buccieri, D., Sanchez, J., Dormido, S., Mullhaupt, P. & Bonvin, D. (2005), Interactive 3d simulation of flat systems: The spidercrane as a case study, *in* 'Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on', pp. 6222 – 6227.

Burns, A. & Wellings, A. (2001), *RealTime Systems and Programming Languages*, 3rd edn, Addison–Wesley.

Buttazzo, G. & Kuo, T. (2009), 'Guest editorial: Special issue on real-time systemspart ii', *IEEE Transactions on Industrial Informatics* **5**(1), 1–2.

Calerga (2010), 'Sysquake home page'.
URL: *www.calerga.com/products/Sysquake/*

Cellier, F. (1991), *Computer System Modeling*, Springer–Verlag.

Cervin, A. (2003), Integrated Control and Real-Time Scheduling, PhD thesis, Lund Institute of Technology.

Cervin, A., Henriksson, D., Lincoln, B., Eker, J. & Årzén, K. (2003), 'How does control timing affect performance?', *IEEE Control Systems Magazine* **23**(3), 16–30.

Chandrasekara, C. & Davari, A. (2004), Inverted pendulum: an experiment for control laboratory, *in* 'Proceedings of the Thirty-Sixth Southeastern Symposium on System Theory', pp. 570 – 573.

Christian, W. (2007), *Open Source Physics: A User's Guide with Examples*, Pearson Education.

Christian, W. (2010), 'Open source physics' home page'.
URL: *www.opensourcephysics.org*

Cristea, S., de Prada, C. & Keyser, R. D. (2005), Predictive control of a process with variable dead-time, *in* 'Proceedings of the IFAC 16th World Congress'.

Demuth, H., Beale, M. & Hagan, M. (2009), *Neural Network Toolbox 6, User's Guide*, The Mathworks.

Department of Computer Science and Automatic Control, UNED (2010*a*), 'JIMC's home page'.
URL: *lab.dia.uned.es/rmatlab*

Department of Computer Science and Automatic Control, UNED (2010*b*), 'JIM's home page'.
URL: *lab.dia.uned.es/rmatlab*

Department of Computer Science and Automatic Control, UNED (2010*c*), 'JTT's home page'.
URL: *lab.dia.uned.es/jtt*

Dong-Jin, L. (2006), 'A laboratory course in real–time software for the control of dynamic systems', *IEEE Transactions on Education* **49**(3), 346–354.

Dorf, R. & Bishop, R. (2004), *Modern Control Systems*, 10th edn, Prentice Hall.

Dormido-Canto, S., Farias, G., Dormido, R., Sánchez, J., Duro, N., Vargas, H., Vega, J., Ratta, G., Pereira, A. & Portas, A. (2008), 'Structural pattern recognition methods based on string comparison for fusion database', *Fusion Engineering and Design* **83**(2–3), 421–424.

Dormido-Canto, S., Farias, G., Vega, J., Dormido, R., Sánchez, J., Duro, N., Vargas, H., Murari, A. & Contributors, J.-E. (2008), 'Classifier based on support vector machine for jet plasma configurations', *Review of Scientific Instruments* **79**, 10F326–1/10F326–3.

Dormido, R., Vargas, H., Duro, N., Sánchez, J., Dormido-Canto, S., Farias, G., Esquembre, F. & Dormido, S. (2008), 'Development of a web-based control laboratory for automation technicians: The three-tank system', *IEEE Transactions on Education* **51**(1), 35–44.

Dormido, S. (2002), 'Control learning: Present and future', *IFAC Annual Control Reviews* **28**, 115–136.

Dormido, S., Dormido-Canto, S., Dormido, R., Sánchez, J. & Duro, N. (2005), 'The role of interactivity in control learning', *The International Journal of Engineering Education: Especial issue on Control Engineering Education* **21**(6), 1122–1133.

Dormido, S. & Esquembre, F. (2003), The quadruple–tank process: An interactive tool for control education, *in* 'Proceedings of the European Control Conference'.

Dormido, S., Esquembre, F., Farias, G. & Sánchez, J. (2005), Adding interactivity to existing simulink models using easy java simulations, *in* 'Proceedings 44th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC'05)', pp. 4163–4168.

Dormido, S., Martín, C., Pastor, R., Sánchez, J. & Esquembre, F. (2004), Magnetic levitation system: A virtual lab in easy java simulation, *in* 'Proceedings of the American Control Conference'.

Duda, R., Hort, P. & Stork, D. (2001), *Pattern Classification*, 2nd edn, Wiley–Interscience.

Duro, N., Dormido, R., Vargas, H., Dormido-Canto, S., Sánchez, J., Farias, G., Esquembre, F. & Dormido, S. (2008), 'An integrated virtual and remote control lab: The three-tank system as a case study', *Computing in Science and Engineering Magazine* **10**(4), 50–58.

Eaton, J. (2002), *GNU Octave Manual*, Network Theory Limited.

Eaton, J. (2010), 'Octave home page'.
URL: *www.gnu.org/software/octave/*

Elmqvist, H., Mattsson, S. & Otter, M. (1998), MODELICA – the new object–oriented modelling language, *in* 'Proceedings of the 12th European Simulation Multiconference'.

Esquembre, F. (2004), 'Easy java simulations: A software tool to create scientific simulations in java', *Computer Physics Communications* **156**, 199–204.

Esquembre, F. (2005), *Creación de Simulaciones Interactivas en Java*, Pearson Educación.

Esquembre, F. (2010), 'Easy Java Simulations home page'.
URL: *www.fem.um.es/Ejs*

Esquembre, F., Dormido, S. & Farias, G. (2007), Defining and performing experiments in virtual laboratories, *in* 'Proceedings of the 10th International Conference on Engineering Education, ICEE2007'.

Fabregas, E., Farias, G., Dormido-Canto, S., Dormido, S. & Esquembre, F. (2010), 'A practical approach for remote interaction with a real plant', *Submitted to Computer & Education* .

Faison, T. (2006), *Event-Based Programming: Taking Events to the Limit*, Springer-verlag.

Farias, G., Årzén, K. & Cervin, A. (2007), Interactive real-time control labs with true-time and easy java simulations, *in* 'Proceedings of the International Multiconference on Computer Science and Information Technology, International Workshop on Real Time Software, IMCSIT2007', pp. 811–820.

Farias, G., Årzén, K., Cervin, A., Dormido, S. & Esquembre, F. (2009), 'Teaching embedded control systems', *International Journal of Engineering Education* . Submitted.

Farias, G., Cervin, A., Årzén, K., Dormido, S. & Esquembre, F. (2008), Multitasking real-time control systems in easy java simulations, *in* 'Proceedings of the 17th IFAC World Congress'.

Farias, G., Cervin, A., Årzén, K., Dormido, S. & Esquembre, F. (2009), 'Java simulations of embedded control systems', *Real-Time Systems* . Submitted.

Farias, G., Dormido-Canto, S., Vega, J., Sánchez, J., Duro, N., Dormido, R., Ochando, M., Santos, M. & Pajares, G. (2006), 'Searching for patterns in tj-ii time evolution signals', *Fusion Engineering and Design* **81**, 1993–1997.

Farias, G., Dormido, R., Santos, M. & Duro, N. (2005), 'Image classifier for the tj-ii thomson scattering diagnostic: Evaluation with a feed forward neural network', *Lecture Notes in Computer Science* **3562**(2), 604–612.

Farias, G., Dormido, S., Esquembre, F., Vargas, H. & Dormido-Canto, S. (2008), Laboratorio virtual para la enseñanza de técnicas de reconocimiento de patrones, *in* 'Proceedings of the 13th Latin-American Congress on Automatic Control'.

Farias, G., Esquembre, F., Sánchez, J., Dormido, S., Vargas, H., Dormido-Canto, S., Dormido, R. & Duro, N. (2006*a*), Desarrollo de laboratorios virtuales, interactivos y remotos utilizando easy java simulations y modelos simulink, *in* 'Proceedings of the 12th Latin-American Congress on Automatic Control', pp. 336–341.

Farias, G., Esquembre, F., Sánchez, J., Dormido, S., Vargas, H., Dormido-Canto, S., Dormido, R. & Duro, N. (2006*b*), Laboratorios virtuales remotos usando easy java simulations y simulink, *in* 'XXVII Jornadas de Automática', pp. 926–933.

Farias, G., Keyser, R. D., Dormido, S. & Esquembre, F. (2009), Building remote labs using easy java simulation and matlab, *in* 'Proceedings of the 10th European Control Conference'.

Farias, G., Keyser, R. D., Dormido, S. & Esquembre, F. (2010), 'Developing networked control labs: A matlab and easy java simulations approach', *IEEE Transactions on Industrial Electronics* . DOI:10.1109/TIE.2010.2041130.

Farias, G., Santos, M. & Dormido-Canto, S. (2005), Desarrollo de una aplicación para la integración de técnicas de reconocimiento de patrones, *in* 'XXVI Jornadas de Automática'.

Farias, G., Santos, M. & J. Marrón, S. D.-C. (n.d.), Determinación de parámetros de la transformada wavelets para la clasificación de señales del diagnóstico scattering thomson, *in* 'XXV Jornadas de Automática'.

Fritzson, O., Gunnarsson, J. & Jirstand, M. (2002), Mathmodelica. an extensible modeling and simulation environment with integrated graphics and literate programming, *in* 'Proceedings of the 2nd International Modelica Conference'.

Gelb, A. & Velde, W. V. (1968), *Multiple-Input Describing Functions and Nonlinear System Design*, McGraw-Hill.

Gillet, D., El Helou, S., Yu, C. M. & Salzmann, C. (2008), Turning web 2.0 social software into versatile collaborative learning solutions, *in* 'Advances in Computer-Human Interaction', pp. 170 –176.

Gillet, D., Ngoc, A. V. N. & Rekik, Y. (2005), 'Collaborative web-based experimentation in flexible engineering education', *Education, IEEE Transactions on* **48**(4), 696 – 704.

Gillet, D., Salzmann, C., Latchman, H. & Crisalle, O. (2000), Recent advances in remote experimentation, *in* 'Proceedings of the American Control Conference', pp. 2955 – 2956.

Gomes, L. & Bogosyan, S. (2009), 'Current trends in remote laboratories', *IEEE Transactions on Industrial Electronics* **56**(12), 4744–4756.

Gomes, L., Coito, F., Costa, A., Palma, L. & Almeida, P. (2007), Remote laboratories support within teaching and learning activities, *in* 'International Conference on Remote Engineering and Virtual Instrumentation (REV'2007)'.

Guzmán, J. (2006), Interactive Control System Design, PhD thesis, Almería University.

Guzmán, J., Rodríguez, F., Berenguel, M. & Dormido, S. (2005), Virtual lab for teaching greenhouse climate control, *in* '16th IFAC World Congress'.

Heck, B. (1999), 'Special report: Future directions in control education', *IEEE Control Systems Magazine* **19**(5), 35–58.

Hernandez, A., Maanas, M. & Costa-Castello, R. (2008), 'Learning respiratory system function in bme studies by means of a virtual laboratory: Respilab', *Education, IEEE Transactions on* **51**(1), 24 –34.

Hilera, J. & Martínez, V. (1995), *Redes Neuronales Artificiales. Fundamentos, modelos y aplicaciones*, Rama.

IEEE (1990), *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, IEEE.

Internacional, E. (2010), 'EcosimPro's home page'.
URL: *www.ecosimpro.com/*

Karayanakis, N. (1995), *Advanced System Modelling and Simulation with Block Diagram Languages*, CRC Press.

Kurose, J. & Ross, K. (2009), *Computer Networking: A top-down approach*, 5th edn, Pearson Education.

Latchman, H., Salzmann, C., Gillet, D. & Bouzekri, H. (1999), 'Information technology enhanced learning in distance and conventional education', *Education, IEEE Transactions on* **42**(4), 247 –254.

Latchman, H., Salzmann, C., Gillet, D. & Kim, J. (2001), 'Learning on demand-a hybrid synchronous/asynchronous approach', *Education, IEEE Transactions on* **44**(2), 17 pp.

Lazar, C. & Carari, S. (2008), 'A remote-control engineering laboratory', *IEEE Transactions on Industrial Electronics* **55**(6), 2368–2375.

Leva, A. (2003), 'A hands-on experimental laboratory for undergraduate courses in automatic control', *Education, IEEE Transactions on* **46**(2), 263 – 272.

Leva, A. (2004), An experimental laboratory on control structures, *in* 'American Control Conference, 2004. Proceedings of the 2004', Vol. 4, pp. 3227 – 3232.

Leva, A. (2006), Real-time web-based interactive control experiments with fast sampling times, *in* 'Proceedings of the Advances in Control Education (ACE)'.

Leva, A. & Donida, F. (2008), 'Multifunctional remote laboratory for education in automatic control: The crautolab experience', *Industrial Electronics, IEEE Transactions on* **55**(6), 2376 –2385.

Liang, S. (1999), *The Java Native Interface, programmer's guide and specification*, Addison–Wesley.

Liu, G., Xia, Y., Chen, J., Rees, D. & Hu, W. (2007), 'Networked predictive control of systems with random network delays in both forward and feedback channels', *IEEE Trans. Ind. Electron.* **54**(3), 1282–1297.

Lund University (2010), 'TrueTime's home page'.
URL: *www.control.lth.se/truetime*

M. Ohlin, D. H. & Cervin, A. (2007), *TrueTime 1.5 Reference Manual*, Department of Automatic Control.

Mallat, S. (2001), *A Wavelet Tour of signal Processing*, 2nd edn, Academia Press.

Martín, C., Urquía, A., Sánchez, J., Dormido, S., Esquembre, F., Guzmán, J. & Berenguel, M. (2004), Interactive simulation of object-oriented hybrid models, by combined use of ejs, matlab/simulink and modelica/dymola, *in* 'Proceedings of the 18th European Simulation Multiconference'.

Misiti, M., Misiti, Y., Oppenheim, G. & Poggi, J. (2009), *Wavelet Toolbox 4, User's Guide*, The Mathworks.

Müller, S. (2010), 'JMatLink's home page'.
URL: *www.held-mueller.de/JMatLink/*

Nolte, T. & Passerone, R. (2009), 'Guest editorial special section on real-time and (networked) embedded systems', *IEEE Transactions on Industrial Informatics* **5**(3), 198–201.

Normey-Rico, J. & Camacho, E. (2007), *Control of Dead-time Processes*, Springer.

Obaidat, M. & Papadimitriou, G. (2003), *Applied system simulation: methodologies and applications*, Kluwer Academic Publishers.

Ogata, K. (2006), *Modern Control Engineering*, international edn, Prentice Hall.

Oppenheim, A., Willsky, A. & Hamid, S. (1997), *Signals and Systems*, 2nd edn, Prentice Hall.

Perritaz, D., Salzmann, C. & Gillet, D. (2009), Quality of experience for adaptation in augmented reality, *in* 'Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on', pp. 888 –893.

Piguet, Y. & Gillet, D. (1999), Java-based remote experimentation for control algorithms prototyping, *in* 'American Control Conference, 1999. Proceedings of the 1999', pp. 1465 –1469.

Radharamanan, R. & Jenkins, H. (2007), Robot applications in laboratory-learning environment, *in* 'Proceedings of the Thirty-Ninth Southeastern Symposium on System Theory', pp. 80 –84.

Salzmann, C., Gillet, D. & Müllhaupt, P. (2005), Real-time interaction over the internet: Model for qos adaptation, *in* 'Proceedings of the IFAC 16th World Congress'.

Sánchez, J. (2001), Un nuevo enfoque metodológico para la enseñanza a distancia de asignaturas experimentales. Análisis, diseño y desarrollo de un laboratorio virtual y remoto para el estudio de la automática a través de Internet, PhD thesis, Universidad Nacional de Educación a Distancia (UNED).

Sánchez, J., Dormido-Canto, S., Farias, G., Dormido, S. & Godoy, F. (2010), 'Understanding automatic control concepts by playing games', *Submitted to International Journal of Engineering Education* .

Sánchez, J., Dormido, S. & Esquembre, F. (2005), 'The learning of control concepts using interactive tools', *Computer Applications in Engineering Education* **13**(1), 84–98.

Sánchez, J., Dormido, S., Pastor, R. & Esquembre, F. (2004), Interactive learning of control concepts using easy java simulations, *in* 'Proceedings of the IFAC Workshop Internet Based Control Education IBCE'04'.

Sánchez, J., Morilla, F., Dormido, S., Aranda, J. & Ruiperez, P. (2002), 'Virtual and remote control labs using java: a qualitative approach', *Control Systems Magazine, IEEE* **22**(2), 8 –20.

Scilab Consortium (2010), 'Scilab home page'.

URL: *www.scilab.org*

SIIA (2001), Trends report 2001: Trends shaping the digital economy, Technical report, The software & Information Industry Association.

Software, M. (1995), *ACSL Reference Manual, version 11*, MGA Software.

Spong, M. & Block, D. (1995), The pendubot: a mechatronic system for control research and education, *in* 'Proceedings of the 34th IEEE Conference on Decision and Control', pp. 555 –556.

The MathWorks (2010), 'MATLAB home page'.

URL: *www.mathworks.com*

The MathWorks, I. (2007), *Data Acquisition toolbox, user's guide*, The MathWorks, Inc.

The MathWorks, I. (2009*a*), *MATLAB External Interfaces*, The MathWorks, Inc.

The MathWorks, I. (2009*b*), *MATLAB Getting Started Guide*, The MathWorks, Inc.

The MathWorks, I. (2009*c*), *Simulink User's Guide*, The MathWorks, Inc.

Uran, S. & Jezernik, K. (2008), 'Virtual laboratory for creative control design experiments', *Education, IEEE Transactions on* **51**(1), 69 –75.

Vargas, H., Dormido, R., Duro, N., Sánchez, J., Dormido-Canto, S., Farias, G., Dormido, S. & Esquembre, F. (2006), Heatflow: Un laboratorio basado en web usando easy java simulations y labview para el entrenamiento de técnicas de automatización, *in* 'Proceedings of the 12th Latin-American Congress on Automatic Control', pp. 330–335.

Vargas, H., Sánchez, J., Duro, N., Dormido, R., Dormido-Canto, S., Farias, G., Dormido, S., Esquembre, F., Salzmann, C. & Gillet, D. (2008), 'A systematic two-layer approach to develop web-based experimentation environments for control engineering education', *Intelligent Automation and Soft Computing* **14**(4), 505–524.

Vega, J., Pastor, I., Cereceda, J., Pereira, A., Herranz, J., Pérez, D., Rodríguez, M., Farias, G., Dormido-Canto, S., Sánchez, J., Dormido, R., Duro, N. & Dormido, S.

(2005), Application of intelligent classification techniques to the tj-ii thomson scattering diagnostic, *in* '32nd EPS Plasma Physics Conference, 8th International Workshop on Fast Ignition of Fusion Targets'.

Vormoor, O. (2001), 'Quick and easy interactive molecular dynamics using java3d', *Computing in Science Engineering* **3**(5), 98 –104.

Wellstead, P. E. (1983), 'The ball and hoop system', *Automatica* **19**(4), 401 – 406.

Wu, M., She, J.-H., Zeng, G.-X. & Ohyama, Y. (2008), 'Internet-based teaching and experiment system for control engineering course', *Industrial Electronics, IEEE Transactions on* **55**(6), 2386 –2396.

Yen, C., Li, W.-J. & Lin, J.-C. (2003), 'A web-based, collaborative, computer-aided sequential control design tool', *Control Systems Magazine, IEEE* **23**(2), 14 – 19.

Appendix A

Modifying a Simulink Model to Control it from MATLAB

A.1 General modifications of Simulink models

Controlling Simulink from MATLAB can be done programmatically by using the API that Simulink provides. The API allows users to open, simulate, pause, stop, or close a Simulink model. It is also possible, to add or remove blocks from the Simulink model, and to modify any block parameter while the simulation is stopped or even while it is running.

Table A.1: Some common Simulink functions.

Function	Description
open_system	Open existing model or block.
close_system	Close open model or block.
add_block	Add new block.
delete_block	Delete a block.
add_line	Add a line.
delete_line	Remove a line.
set_param	Set parameter values for model or block.
get_param	Get simulation parameter values from model.

Table A.1 shows some common functions of the Simulink API. The functions `open` and `close` open and close a Simulink model, respectively. The model can be modified by adding or deleting blocks with `add_block` and `delete_block`. Functions `add_line` and `delete_line` allow to connect or disconnect blocks.

The function `set_param` modifies a parameter of a block or model, which can strongly affect the behaviour when simulated. This function is used as follows:

```
set_param('blockpath','parameter','value');
```

The `'blockpath'` string defines the path to the block. The path represents the route

to the block inside the model. The `parameter` string indicates the parameters that will be modified, and the `value` string is the new value of the modified parameter.

There are some common parameters for the blocks and models, but obviously different blocks can present an important set of different parameters. To get all the parameters of a block, the function `get_param` can be used as follows:

```
params = get_param('blockpath','objectparameters');
```

The `'objectparameters'` string indicates the information of all parameters of the block or model that are requested. A subset of the most used parameters of a block can be obtained by using `'dialogparameters'` instead `'objectparameters'`. This subset of parameters can be modified by using a dialogue box, which appears when users double-click on the block. The obtained `params` variable is of type `struct` where all the fields represent a parameter of the block. The following is a typical content of the variable `params`.

```
params =
                  Name: [1x1 struct]
                   Tag: [1x1 struct]
           Description: [1x1 struct]
                  Type: [1x1 struct]
                Parent: [1x1 struct]
                Handle: [1x1 struct]
      HiliteAncestors: [1x1 struct]
      RequirementInfo: [1x1 struct]
                 Ports: [1x1 struct]
                   ...
```

To obtain information of a parameter such as `Name`, the command `params.Name` has to be executed. Programmers who want to get a complete description of the block parameters should consult the Simulink reference or user guide to learn the correct use of the `set_param` function.

Using the described functions, a programatic control of a Simulink simulation can be performed. This will be exemplified by using a Simulink model named **fsmk.mdl** that evaluates a function similar to the example described using the Java class of `MatlabExternalApp` in Figure 3.3.

The Simulink version of this example is shown in Figure A.1. Here the model (Figure A.1a) uses simple blocks to evaluate the function: $sin(2\pi ft) \cdot cos(t)$.

Note that the variable `f` is given by a block constant named `frequency`. Note also,

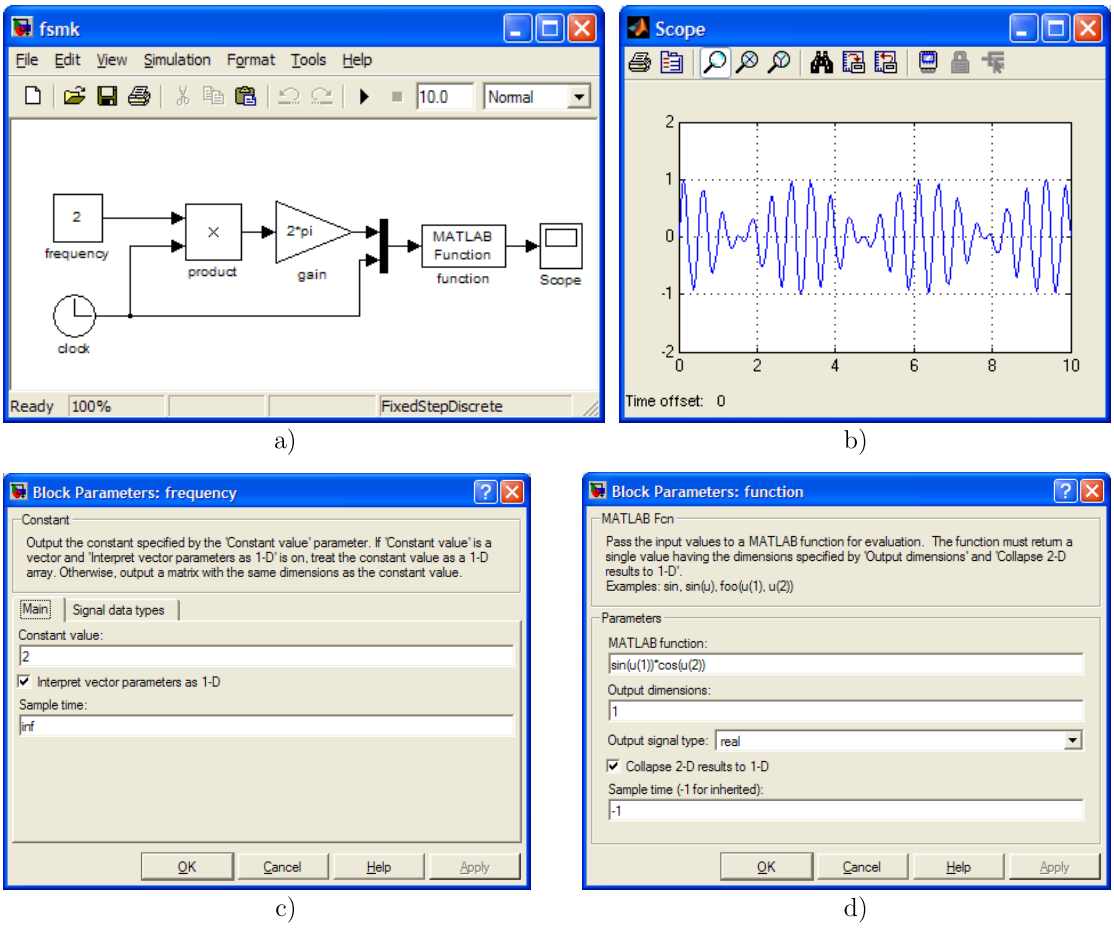


Figure A.1: Simulink version of evaluating function. a) The model, b) a plot of the function, c) some parameters of the block frequency, d) some parameters of the block function.

that the variable t (the time of the simulation) is given by a block clock named `clock`. The output of the frequency (whose value 2 is entered in the dialogue box of Figure A.1c) is multiplied by the output of the block clock, obtaining the value $f \cdot t$ at the output of the block named `product`. The output of the block `product` is amplified by the block gain named `gain`, which feeds with the value $2\pi ft$ the first input of a multiplexor block. The second input of the multiplexor block is fed with the time of the simulation (t).

The block multiplexor sends out both inputs as a vector of two components. This vector has to be treated as an array called u inside the block `function`. The block `function` is a `MATLAB Fcn` block, which can execute any MATLAB function or method at each integration step of the simulation. In this case, the function to be executed by the block `function` is $sin(u(1)) \cdot cos(u(2))$, where $u(1)$ and $u(2)$ are the first and the second components of the u array, respectively.

In Figure A.1d, the MATLAB function is entered. The results of the evaluation of the function is sent to the block `Scope`. This block plots its inputs, in this case

the output of the block `function`. A typical plot drawn by the Scope is presented in Fig.A.1b.

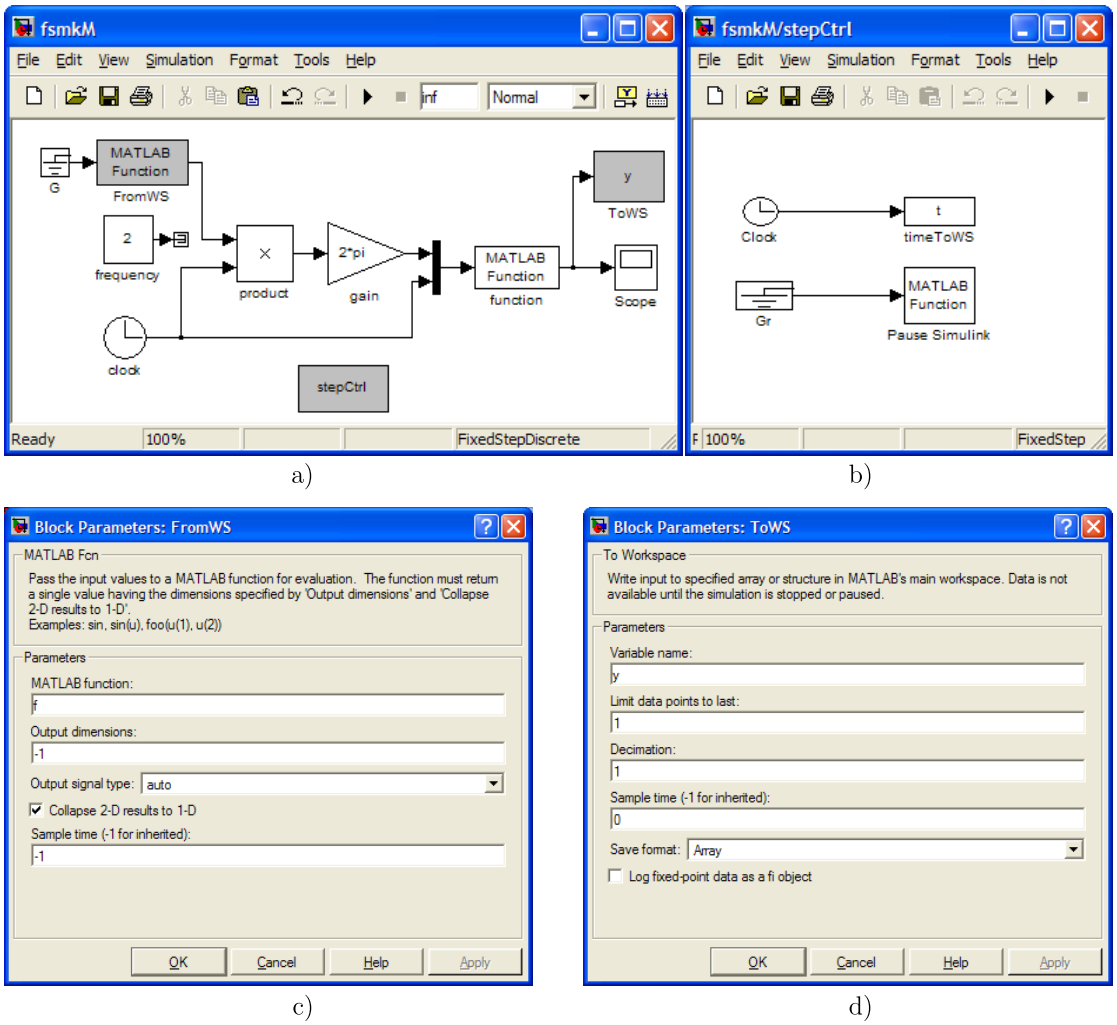


Figure A.2: A modified version of the Simulink model of the Figure A.1a. a) The modified model, b) the submodel `stepCtrl`, c) the parameters for the block `FromWS`, and d) the parameters for the block `ToWS`.

To control this simulation from MATLAB, it is necessary to change the model as the Figure A.2a shows. This modified model, named **fsmkM.mdl**, presents the following modifications:

- The finish time of the Simulink model has to be changed to the value `inf` in order to perform the simulation as long as the Java program requires.
- A block `MATLAB Fcn` named `FromWS` is added to read the value of the variable `f` from the MATLAB workspace. Figure A.2c shows some parameters of this block.
- A block `To Workspace` named `ToWS` is added to write its inputs (the output of the

block `function`) to the MATLAB workspace. Figure A.2d shows some parameters of this block.

- A submodel named `stepCtrl` is added to write the simulation time in the MATLAB workspace and also to pause the model after each integration step. The blocks of this submodel are presented in Figure A.2b. Here, a block `MATLAB Fcn` named `Pause Simulink` pauses the Simulink model by calling the function `set_param`. Note also that another block `MATLAB Fcn` is used to capture the simulation time.

- The rest of the blocks (`Ground`s and `Terminator`s) are added to avoid having blocks with unconnected inputs or outputs.

As mentioned, the function `set_param` can control the simulation of a Simulink model, by manipulating the value of the parameter `'SimulationCommand'` of a model. Some accepted values for this parameter are:`'start'`, `'stop'`, `'pause'` and `'continue'`. These values are used to initiate, stop, pause, or advance one integration step of the simulation.

To advance one integration step, the following command can be used:

```
set_param('fsmkM','SimulationCommand','continue');
```

Obviously the model can be paused using the same function, but with the value `'pause'`, instead.

Another important parameter to control a simulation is `'SimulationStatus'`. This `read-only` parameter indicates the state of the simulation. Some of the values of this parameter are: `'running'`, `'paused'` and `'stopped'`, which means that the model is running, paused, or stopped, respectively.

Furthermore, to check whether the model is paused, the following command could be executed:

```
get_param('fsmkM','SimulationStatus')
```

Which will return the string `'paused'` if the model is paused.

A.1.1 Controlling a modified Simulink model from Java

Now that the main functions to control the Simulink model have been described, the next step is to create an interactive simulation with the Java MatlabExternalApp class. List-

ing A.1 shows the Java code that simulates the modified Simulink model **fsmkM.mdl** using the MatlabExternalApp class.

The code starts declaring the variables and calling the main method of the class `evaluatingFunctionSimulink`. In the method `evaluatingFunctionSimulink()` a MatlabExternalApp instance is first obtained. This object is used to communicate with MATLAB as was explained above. After the connection with MATLAB is started, the Simulink simulation is prepared. To do that, the model is first opened with the MATLAB command `open_system('fsmkM')`. Then, the MATLAB variable `f` is set to the value given by the Java variable `frequency`. Then, the parameter `MATLABfcn` of the block `function` is set to $sin(u(1)) * cos(u(2))$. After that, the simulation is started, with the MATLAB command `set_param('fsmkM','SimulationCommand','start')`. At this moment, the Simulink model is ready to be simulated.

The simulation is run by executing three main action inside of a do-while cycle. The first action, steps the Simulink model, which means that the simulation advances one integration step. After that, it is necessary to check if the execution of this integration step has finished. This checking is done by verifying that the value of the parameter `SimulationStatus` of the model is the `'paused'` string. Take into account that, sooner or later, the Simulink will be paused when the block named `Pause Simulink` is executed. The second action, after the checking, is to get the values of the MATLAB variables `t` and `y`, which represent the time and the output of the `function` block. The third action just prints these values to the console.

The do-while cycle is executed until time is greater than 10. The Simulink model is then stopped and the MATLAB connection finished.

```
1  public class evaluatingFunctionSimulink{
2    //Declare variables
3    public double time=0, frequency=2, value=0;
4    String status;
5
6    public static void main (String[] args) {
7      new evaluatingFunctionSimulink();
8    }
9
10   public evaluatingFunctionSimulink(){
11     //Create a Matlab connection
12     ExternalApp externalApp=new MatlabExternalApp();
13
14     //Start the connection
15     externalApp.connect();
16
17     //Open and prepare simulation
18     externalApp.eval("open_system('fsmkM')");
19     externalApp.setValue("f",frequency);
```

```
20      externalApp.eval("set_param('fsmkM/function',
21                       'MATLABfcn','sin(u(1))*cos(u(2))')");
22      externalApp.eval("set_param('fsmkM','SimulationCommand','start')");
23
24      //Perform the simulation
25      do{
26        //Step the model
27        externalApp.eval("set_param('fsmkM','SimulationCommand','continue')");
28        do{
29          externalApp.eval("s=get_param('fsmkM','SimulationStatus')");
30          status=externalApp.getString("s");
31        }while (!status.equals("paused"));
32
33        //Get variables
34        value=externalApp.getDouble("y");
35        time=externalApp.getDouble("t");
36
37        System.out.println("time:"+time+" value:"+value);
38      } while (time<10);
39
40      //Stop the Simulink simulation
41      externalApp.eval("set_param('fsmkM','SimulationCommand','stop')");
42
43      //Finish the connection
44      externalApp.disconnect();
45    }
46  }
```

Listing A.1: Computing a Function Using a Simulink model.

A.1.2 General process to simulate Simulink models from Java

The process described opens the way to simulating any Simulink model from a Java program. Thus, it is possible to summarize the process required to create interactive simulations using Simulink in the following actions:

- Modify the original Simulink model to indicate that the simulation ends at the time `inf`.
- Modify the original Simulink model, adding blocks to read and write variables from the MATLAB workspace.
- Modify the original Simulink model, adding blocks to obtain the simulation time and also to pause the model by calling the function `set_param` to set the parameter `SimulationCommand` to the string `'pause'`.
- In the Java program, and after the MATLAB connection is started, the modified model has to be opened with the `open_system` function. Then, the variables required by the Simulink model have to be initiated. After that, the model is started setting the parameter `SimulationCommand` of the model to the string `'start'`.

- The simulation is performed, by calling three actions: advancing one integration step of the model, checking that the model has finished the integration process, and recovering the values of the variables than are written by the model to the MATLAB workspace.

- After performing the simulation of the Simulink model, stop the model by setting the parameter `SimulationCommand` to the string `'stop'`.

- Finally, close the MATLAB connection.

In principle, the previous process can be used to simulate any Simulink model from a Java program. This is especially true for static systems like the model **fsmk**. However, the simulation of dynamic systems can be different. In this kind of models, apart from a static description, differential or difference equations may apply, producing time-varying systems. Thus, contrary to static systems, the behaviour of dynamic systems depend on the initial conditions. For this reason, for example, if two identical balls are dropped from different heights (initial conditions), they will stop bouncing at different times.

Hence, an interactive simulation of a dynamic system should be prepared to accept that users can change the initial conditions at any moment. Thus, for example, an interactive simulation of a bouncing ball could be paused by the user, who moves the ball to a different height to restart the simulation from there.

A.2 Specific modifications for integrators blocks

The description of time-varying systems in Simulink can be done in many ways, but the most common method to formulate dynamic systems in Simulink is to add *Integrator* blocks to the model (see Figure A.3).

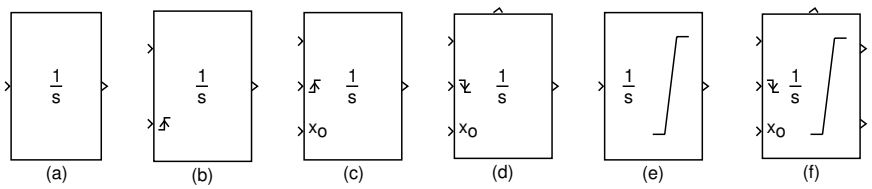


Figure A.3: Various Integrator blocks. The appearance of each Integrator depends on the configuration defined in the dialogue box.

An integrator block outputs the integral of its input at the current time step. Equation (A.1) represents the output of the block y as a function of its input u and an initial

condition y_0, where y and u are vector functions of the current simulation time t. Take into account that the Integrator block outputs the initial condition at the beginning of the simulation and also when the Integrator is reset.

$$y(t) = \int_{t_0}^{t} u(t)dt + y_0 \tag{A.1}$$

Simulink software can use a number of different numerical integration methods to compute the Integrator block's output, each with advantages in particular applications. Users can go to the `Configuration Parameters` dialogue box to select the solver method. Simulink treats the Integrator block as a dynamic system with one state, its output (see Equation (A.2)). The Integrator block's input is the derivative of the state.

$$\begin{aligned} x &= y(t) \\ x_0 &= y_0 \\ \dot{x} &= u(t) \end{aligned} \tag{A.2}$$

The selected solver computes the output of the Integrator block at the current time step, using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current time step which the solver can use to compute its output at the next time step. The block also provides the solver with an initial condition for use while computing the block's initial state at the beginning of a simulation run. The default value of the initial condition is 0. The block's parameter dialogue box (see Figure A.4) allows users to specify another value for the initial condition or create an initial value input port on the block.

The dialogue box of the Integrator also allows users to:

- Define upper and lower limits on the integral.
- Create an input that resets the output of the block (state) to its initial value, depending on how the input changes.
- Create an optional state output so that the value of the block output can trigger

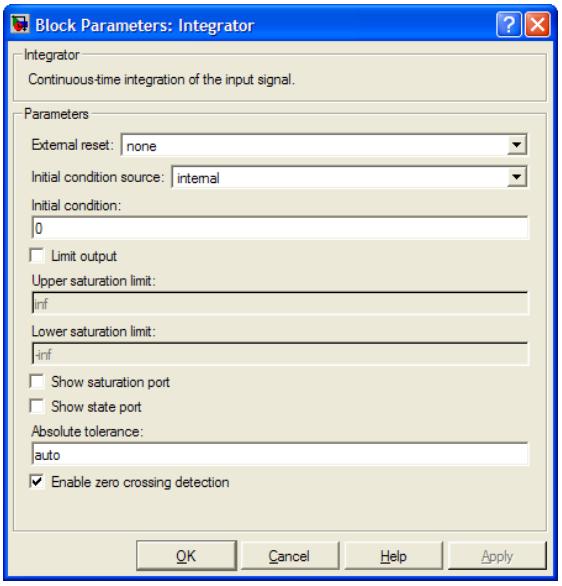


Figure A.4: The parameter dialog box of an Integrator block.

a block reset.

Note that the Integrator block is used to describe continuous systems, but in the case of purely discrete systems, for example in models formulated using difference equations, the suitable integration block is the *Discrete-Time Integrator*, which has parameters similar to the continuous version.

Figure A.3 shows six integrators with different configurations. The first case, Figure A.3a, represents the standard configuration of the Integrator block. The second case, Figure A.3b, shows an extra input for external reset. In this case, the reset is triggered by a rising change of the reset input. The initial condition in this case has to be entered on the corresponding parameter in the dialogue box. Figure A.3c is similar to the previous case, but now the initial condition is taken from the x_0 input. Figure A.3d adds an extra output for the `state port`. The output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. This special output is needed to avoid algebraic loops when the state of the block is required to trigger the reset condition or to feed the input x_0. Note that the reset is triggered by a falling change of the reset input. The fifth case, Figure A.3e, shows an Integrator with a limitation in its standard output. The last case, Figure A.3f, represents an Integrator with external reset and initial condition, state port, limitations of the standard output, and an extra output to indicate, with a 0 or a 1, when the limit conditions are being applied or not.

In order to implement a correct manipulation of the Integrator block for the interaction point of view. This is how to control from Java both the moment when an integrator is reset and the value of the initial condition.

Consider the model (named `integrator`) shown in Fig.A.5, which uses the simplest case of the Integrator block. Since the Integrator block is integrating a constant equal to 1, the output after the integration is a straight line with slope equal to 1. This line is shown in Figure A.6.

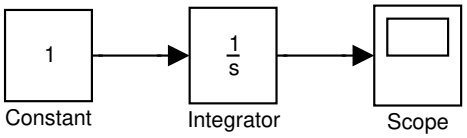


Figure A.5: A model with an integrator.

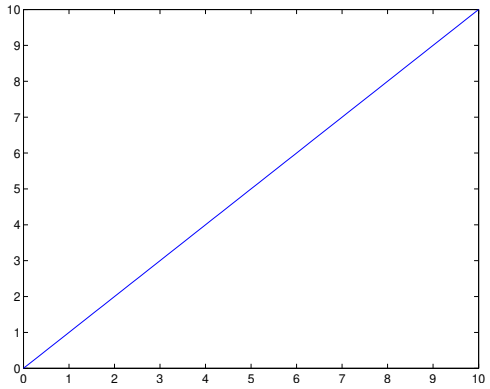


Figure A.6: A plot of the output of the model.

To control the `integrator` model, it needs to be modified to obtain a new model (named `integratorM`) like Figure A.7 shows. Now, the integrator accepts an external reset and external initial condition. The two new inputs of the block integrator are connected to two MATLAB Fcn blocks to manipulate from the MATLAB workspace the reset moment and the initial condition. The two MATLAB Fcn are used to read the variables in the same way as the block named `FromWS` was used in Figure A.2 to read the value of the variable `f`. The two variables read by the blocks MATLAB Fcn are `rst` and `ic`, which control the reset moment and the initial condition respectively. The output of the Integrator is sent to the MATLAB workspace (as variable `y`) using a block `To Workspace` similar to the block `toWS` used in the model of Figure A.2a. Additionally, a sub model similar to the `stepCtrl` shown in Figure A.2b is required to get the simulation time and to pause the Simulink model after each integration step.

After the modifications are done, the model `integratorM` can be used directly from a Java application to reset the integrator. Listing A.2 shows the code of an application that resets the integrator at time=5. Note that the integrator is reset only once, because the reset is only triggered when a rising change in the variable `rst` (e.g., from -1.0 to

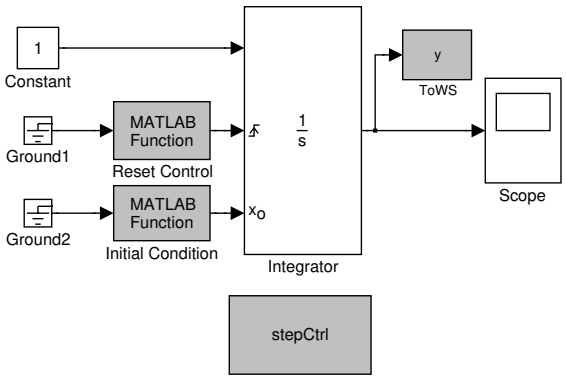


Figure A.7: A modified model of the integrator.

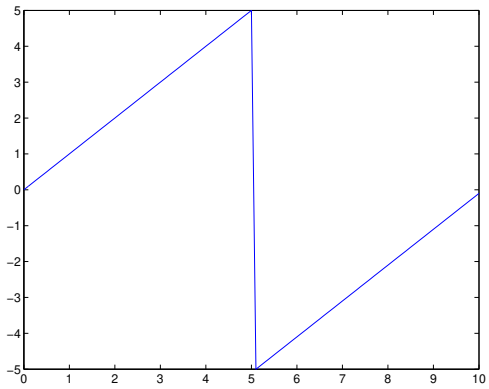


Figure A.8: The output of the modified model.

1.0) is detected. Fig.A.8 shows a plot of the output of the integrator. Observe that, at the beginning of the simulation, the initial condition (`ic`) was 0, and when the reset is triggered, the initial condition is set to -5 and therefore the state restarts from -5.0.

```
...
//Prepare the simulation
externalApp.setValue("rst",-1.0);
externalApp.setValue("ic",0.0);
externalApp.eval("set_param('integratorM','SimulationCommand','start')");
//Perform the simulation
do{
  //Step the model
  externalApp.eval("set_param('integratorM','SimulationCommand','continue')");
  do{
    externalApp.eval("s=get_param('integratorM','SimulationStatus')");
    status=externalApp.getString("s");
  }while (!status.equals("paused"));
  //Get Integrator's output and simulation time
  output=externalApp.getDouble("y");
  time=externalApp.getDouble("t");
  //reset at time=5
  if (time>=5){
    externalApp.setValue("rst",1.0);
    externalApp.setValue("ic",-5.0);
  }
  System.out.println("time:"+time+" output:"+output);
}while (time<10);
...
```

Listing A.2: Resetting an Integrator block from Java.

The simple case of the integrator described previously is however not common. Most of the dynamic systems in Simulink present more complex configurations, which means that sometimes the model itself uses the external reset and external initial condition to model events in the system. To correctly treat the events of the system using the events triggered by Java, the scheme of the Fig.A.9 has to be used.

The scheme requires the substitution of an integrator (with any configuration) by a sub model composed of an integrator with external reset(`Reset`) and external initial condition (`IC`). Both `Reset` and `IC` are computed according to the state of the signals

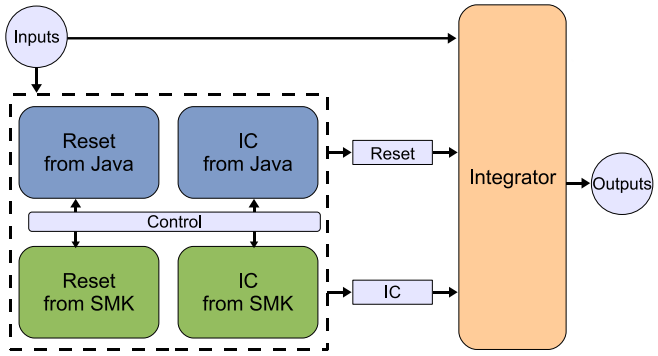


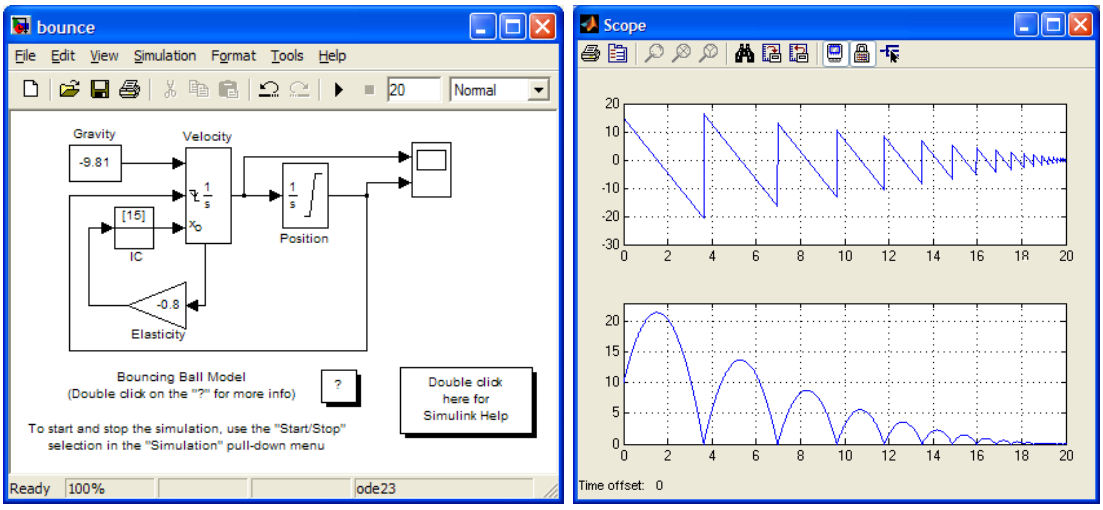
Figure A.9: An scheme to treat Java and Simulink events in integrators.

obtained from Simulink and Java. The Simulink signals reflect the situation of the original integrator in the Simulink model. The Java signals allow manipulating from Java to reset the integrator when, for instance, the user interacts with the Java simulation. Thus, both types of reset have been taken into account in the simulation. In case that both types of reset are triggered at the same type, the Java reset should have priority over the Simulink reset in order to provide a good interaction experience to the end user. Obviously, the configuration of the original integrator such as the limitation and the state port has to be preserved in the new integrator.

A.2.1 An example of a dynamic system

An example of a non elastic bouncing ball will be used to show how the previous ideas are implemented. A rubber ball is thrown into the air with a velocity of 15 meters per second from a height of 10 meters. The position and velocity of the ball are shown in a Scope. This system uses a reset-integrator to change the direction of the ball as it comes into contact with the ground. Figure A.10 shows the Simulink model of the bouncing ball system and the plots generated when the simulation is run.

The model is quite simple. There are two continuous states, the velocity and the position. These states are available as outputs of the two integrators. The position is obtained by integrating the velocity, and the latter is obtained by integrating the gravity given by a Constant block. The integrator for the velocity also has two other inputs, that are used to reset the velocity to an initial state given by the third input. This allows to model the bouncing of the ball when it reaches the ground (i.e., the position is zero). Since the ball is not elastic, the velocity of the ball is reset to a new velocity, which is computed by multiplying the velocity *just before* the bouncing by an elastic



a) b)

Figure A.10: The simulation of a bouncing ball. a) Simulink model, b) Plots of the velocity and vertical position of the ball.

constant equal to −0.8. The negative sign simply changes the direction of the velocity, because just before the bouncing the velocity was pointing down (i.e., negative vertical component). Note that the value of the velocity previous to the bouncing is taken from the state port (the second output) of the block `Velocity` in order to avoid algebraic loops.

The integrator for position has its output limited to a minimum value of zero. This ensures that the position of the ball will never be negative. The Initial Condition block named `IC` outputs 15 when the simulation is started. This is used as the initial condition of the integrator `Velocity`. The Initial Condition block does not have any effect once the simulation is started.

A modified version of the dynamic system

Figure A.11 shows the modified model of the bouncing ball. Since there were two integrators, both have been modified following the scheme shown in Figure A.9. The two new sub models that represent the original integrators `Position` and `Velocity` are now `PositionM` and `VelocityM`, respectively. The necessary submodel `stepCtrl` to get the simulation time and to pause the Simulink model after each integration step has also been added.

The submodel `PositionM` is shown in Figure A.12. The submodel follows the scheme described in Figure A.9. The new integrator has three inputs and one output. One

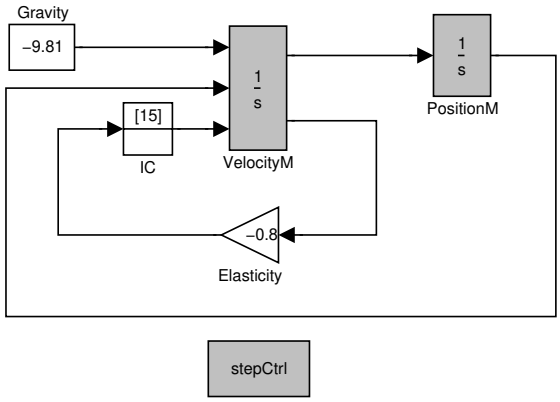


Figure A.11: The modified model of the bouncing ball.

input is used for the standard input of the signal to be integrated, and two inputs for supporting the external reset and external initial condition. The single output is used for the standard output of the integrator, which gives the position of the ball. Note that the position is also written to the MATLAB workspace so that it can be read by the Java application. Observe also that the output limitation of the original integrator is replicated by the new integrator.

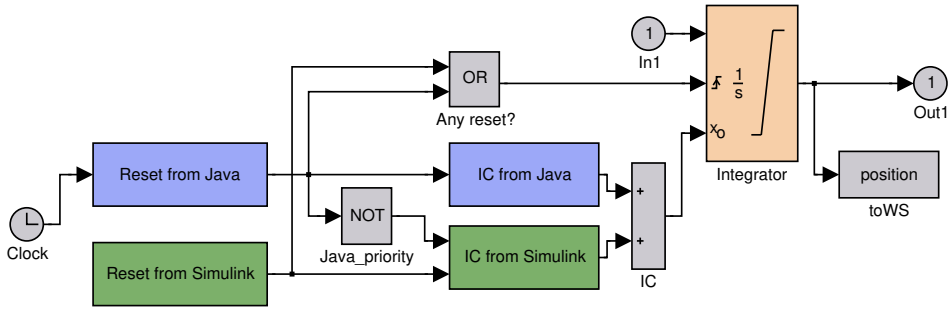


Figure A.12: The sub model `PositionM`.

The standard input of the integrator is taken from the sole input, `In1`, of the submodel. This input is connected to the first output of the submodel `velocityM`, which is the velocity of the ball.

The external reset of the integrator is connected to the block named `Any reset?`, which implements a logic gate OR. This block sends out 1 (or the boolean true) if at least one of its two inputs are 1, otherwise the output is 0 (or the boolean false). The two inputs of the OR block are used to detect any of two possible resets. These inputs come from the submodels: `Reset from Java` and `Reset from Simulink`. If there is a reset, the initial condition is taken from either the submodel `IC from Java` or the submodel

`IC from Simulink`. If both Simulink and Java reset are triggered at the same time, the block NOT, named `Java_priority`, is used to prioritize the Java reset.

The submodel `Reset from Java` is shown in Figure A.13. The reset from Java is triggered switching the value of the variable `rst` from a positive to negative number, and vice versa. The value of the variable is obtained by the block `RS_Java` in the same way as the block `Reset Control` of Figure A.7. When the value of the variable switches, the output of the block named `Reset_Java?` is 1(i.e., true). At the same time, the output of the submodel `Reset from Java` is also 1. Note that at the beginning of the simulation, the output of the submodel is always 1, since there is an IC block named `Resets when it starts` with a value equal to 1.

The submodel `Reset from Simulink` is much simpler than the previous one (see Figure A.14). In this case, since the original integrator for the position did not have an external reset, the output of the submodel is always 0.

If the Java reset is triggered, the submodel `IC from Java` simply sends out the value of the variable `ic_position`, which is read by the block `IC_Java` (see Figure A.15). Otherwise, the output of the submodel is 0.

If the Simulink reset is triggered and there is no Java reset, the submodel `IC from Simulink`(shown in the Fig. see Figure A.16) sends out the initial condition (10) specified in the dialogue box of original integrator `Position`. Otherwise, the output of the submodel is 0.

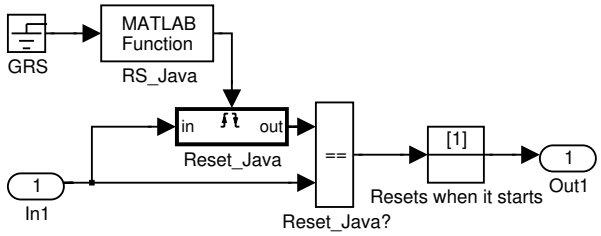


Figure A.13: Submodel `Reset from Java`.

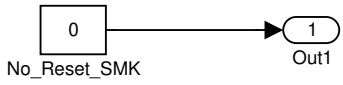


Figure A.14: Submodel `Reset from Simulink`.

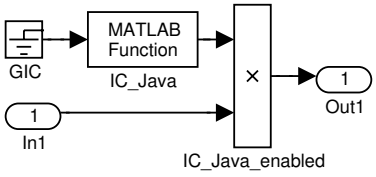


Figure A.15: Submodel `IC from Java`.

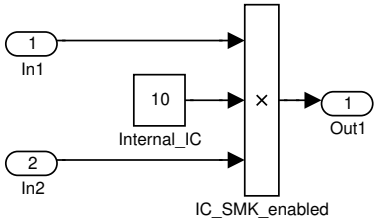


Figure A.16: Submodel `IC from Simulink`.

The submodel `VelocityM` is shown in Figure A.17. The submodel follows the scheme described in Figure A.9 and explained above for the integrator `Position`. The new integrator has the three inputs and two outputs. As before, one input is used for the standard input of the signal to be integrated, and two inputs are used for supporting the external reset and external initial condition. One output is used for the standard output of the integrator, which sends out the velocity of the ball. The second output sends out the state port signal, which is used to compute the velocity when the ball comes into contact with the ground. As before, the velocity is also written in the MATLAB workspace to be read from the Java application.

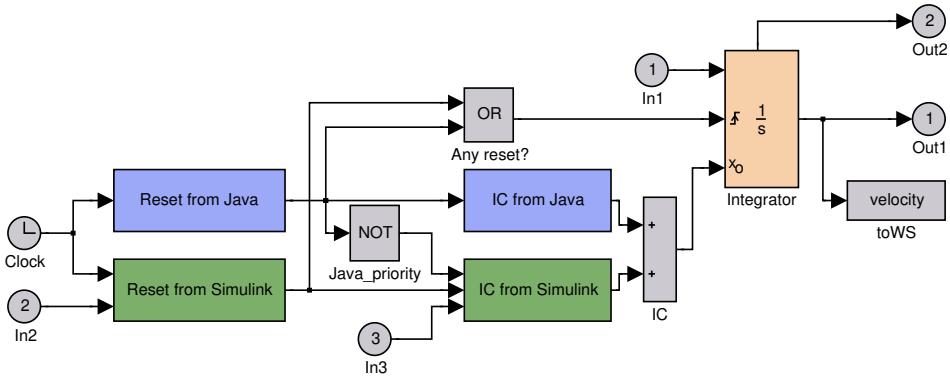


Figure A.17: The submodel `VelocityM`.

The standard input of the integrator is taken from the first input, named `In1`, of the submodel. This input comes from the output of the Constant block, named `Gravity`, which provides the value -9.81 for the gravity. The external reset and the external initial condition of the new integrator are computed as the previous integrator `PositionM`.

The submodel `Reset from Java` of the `VelocityM` is the same used in the submodel `PositionM` shown in Figure A.13. The case of the submodel `IC from Java` is similar, but here the variable to set the initial condition from Java is named `ic_velocity`.

The submodel `Reset from Simulink` (see Figure A.18) is now different from the case in the `PositionM`. Here, the external reset of the original integrator velocity is taken into account. The original external reset signal comes from the second input of the submodel `VelocityM`, this signal is also the second input of the `Reset from Simulink` block. To capture the same behaviour of the original reset, the trigger of the block `Reset_SMK` has to be the same (i.e., rising or falling) as the original trigger of the integrator `Velocity`. Thus, when the original external reset triggers `Reset_SMK` its input is output to the block

`Reset_SMK?`, implying that the output of the submodel `Reset from Simulink` is 1.

As in the submodel `PositionM`, if the Simulink reset is triggered and there is no Java reset, the submodel `IC from Simulink`(shown in the Fig. see Figure A.19) sends out the initial condition. However, in this case, this initial condition is taken from the third input of the submodel `VelocityM` in order to replicate the behaviour of the original integrator `Velocity`. This third input of `VelocityM` is connected to the output of the Initial Condition block `IC` as the Figure A.11 shows.

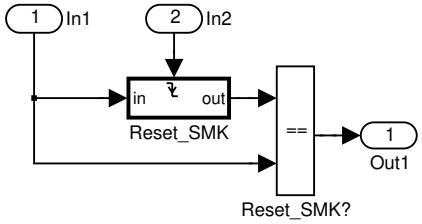


Figure A.18: Submodel `Reset from Simulink`.

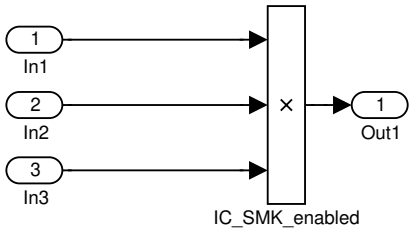


Figure A.19: Submodel `IC from Simulink`.