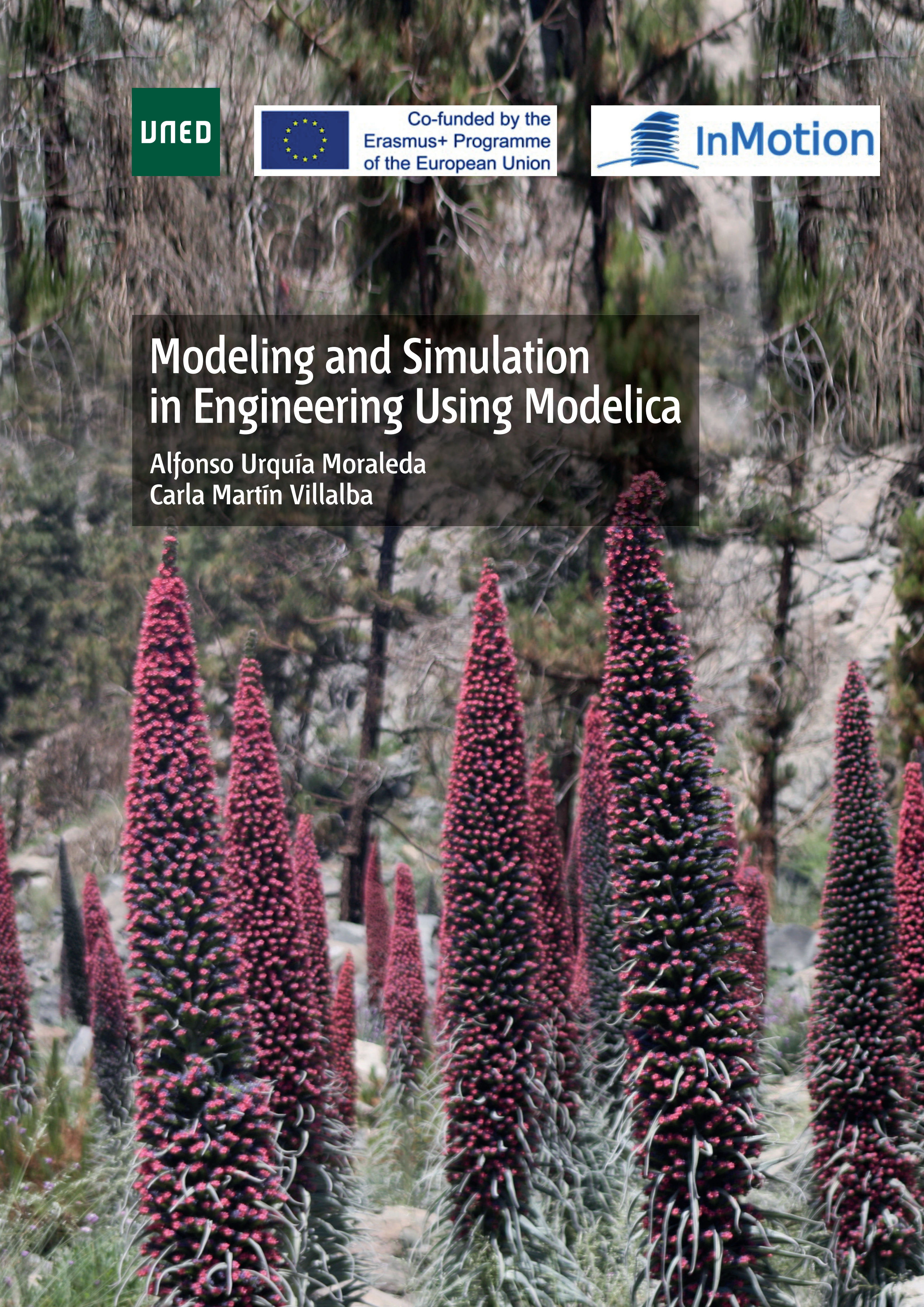


Co-funded by the Erasmus+ Programme of the European Union

Modeling and Simulation in Engineering Using Modelica

Alfonso Urquía Moraleda
Carla Martín Villalba

Modeling and simulation in Engineering Using Modelica

ALFONSO URQUÍA MORALEDA
CARLA MARTÍN VILLALBA

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Co-funded by the
Erasmus+ Programme
of the European Union

InMotion

This publication was conducted within InMotion project (Innovative teaching and learning strategies in open modelling and simulation environment for student-centered engineering education (573751-EPP-1-2016-1-DE-EPPKA2-CBHE-JP)).

MODELING AND SIMULATION IN ENGINEERING USING MODELICA

ISBN electrónico: 978-84-362-7309-0

Edición digital: abril de 2018

Contents

Modelica code

Preface

I Continuous-time modeling

Lesson 1 Modeling methodology and tools

1.1 Introduction
1.2 Physical modeling paradigm
1.3 Object-oriented modeling
1.4 Modeling environments
1.5 Getting started with Modelica
1.6 Further reading

Lesson 2 Continuous-time atomic models

2.1 Introduction
2.2 Rectifier circuit
2.3 Translation in one dimension
2.4 Translation in two dimensions
2.5 Radial heat transfer in a pipe.
2.6 Further reading

Lesson 3 Model libraries

3.1 Introduction
3.2 Electrical library
3.3 Longitudinal vibrations of a bar
3.4 Longitudinal heat conduction in a bar
3.5 Control of level and temperature in a tank
3.6 Dissipation of heat generated in a circuit
3.7 Further reading

II Simulation of continuous-time models

Lesson 4 Computational causality

4.1 Introduction
4.2 Classification of the model variables
4.3 Structural singularity
4.4 Partition algorithm
4.5 Overdetermined and underdetermined systems
4.6 Example: simulation of an electrical circuit
4.7 Further reading

Lesson 5 Index and initialization of DAE systems

5.1 Introduction
5.2 Structurally singular DAE systems
5.3 Index of DAE systems
5.4 Initialization of DAE systems
5.5 Selection of the state variables
5.6 Further reading

Lesson 6 Numerical methods

6.1 Introduction
6.2 Systems of simultaneous equations
6.3 Numerical solution of ODE
6.4 Numerical solution of DAE
6.5 Further reading

III Hybrid system modeling and simulation

Lesson 7 Hybrid system specification

7.1 Introduction
7.2 The OHM formalism
7.3 Model specification and simulation algorithm
7.4 Model specification and Modelica description
7.5 Models with a variable structure
7.6 Model initialization
7.7 Further reading

Lesson 8 Event detection and handling

8.1 Introduction
8.2 Simultaneous events
8.3 Crossing function
8.4 Determination of the event instant
8.5 Chattering
8.6 Further reading

Lesson 9 Hybrid modeling practice
9.1 Introduction
9.2 Ideal electric switch
9.3 Ideal diode
9.4 Two-tank and valve system
9.5 Bouncing ball
9.6 Dry fiction
9.7 Heat conduction in a wall
9.8 Further reading

Subject Index

References

Modelica code

1.1. Model of a simple plane pendulum
2.1. Circuit of Figure 2.1, with the parameters of Table 2.1.
2.2. Fragment of the Modelica.SIunits package
2.3. Circuit of Figure 2.1, described using the Modelica.SIunits types
2.4. Model of the rectifier circuit shown in Figure 2.5.
2.5. Model of the two-body system shown in Figure 2.7.
2.6. Model of the system depicted in Figure 2.9.
2.7. Model of the Sun-Earth system
2.8. Radial heat transfer in an insulated steel pipe
2.9. Linear interpolation funcion and its invocation
3.1. Electrical library (1/3)
3.2. Electrical library (2/3)
3.3. Electrical library (3/3)
3.4. Longitudinal vibration of a uniform bar
3.5. Atomic model of the longitudinal heat conduction in a bar
3.6. Thermal library to model the heat conduction in a bar
3.7. Composed model of the longitudinal heat conduction in a bar
3.8. Tank with level control shown in Figure 3.8.
3.9. Atomic model of level and temperature control in a tank (1/2)
3.10. Atomic model of level and temperature control in a tank (2/2)
3.11. Composed model of level and temperature control in a tank (1/3)
3.12. Composed model of level and temperature control in a tank (2/3)
3.13. Composed model of level and temperature control in a tank (3/3)
5.1. Two-tank and source system shown in Figure 5.1.
5.2. Simple pendulum using Cartesian coordinates
7.1. Initialization of a continuous-time model
7.2. Three different ways of initializing the pendulum model
8.1. Model used to illustrate the simultaneous triggering of two events
8.2. Model to illustrate the event detection in Dymola
8.3. Vertical fall and bouncing of a ball
8.4. Model with chattering of the vertical fall and bouncing of a ball
9.1. Model of the electric circuit shown in Figure 9.2.
9.2. Rectifier circuit shown in Figure 9.5 with ideal diode

9.3. Model of the two-tank and valve system shown in Figure 9.9.
9.4. Bouncing ball model shown in Figure 9.11.
9.5. *LibFriction* library (1/3)
9.6. *LibFriction* library (2/3)
9.7. *LibFriction* library (3/3)
9.8. Heat transfer in the three-layer wall shown in Figure 9.16.
9.9. Heat transfer in the wall, with the B material divided into layers

Preface

The important advances made in the fields of computer hardware and numerical methods in the 1980s paved the way for the development of general-purpose (i.e., not tied to any specific physical domain), equation-based, object-oriented modeling languages in the early 1990s. These languages were intended to facilitate the description of physical system models, where phenomena in different physical domains (e.g., electrical, mechanical, thermo-fluid and chemical) appear interrelated. The target models, the so-called hybrid DAE models, were dynamic mathematical models described in terms of ordinary differential equations with derivative with respect to time, algebraic equations, and events.

In those first years, the use of object-oriented modeling languages was restricted to some academic groups, mainly in the field of Control Engineering. The coexistence of a plethora of modeling languages led to dispersion of the efforts in the development of software tools and model repositories. Similar model libraries were programmed by different developers from scratch, being unfeasible to reuse code previously made by others because the models were written in different languages. Likewise, as dedicated modeling environments (i.e., software tools for editing and simulating models) were developed for each modeling language, improvements in a software tool were not easily applicable to other tools and only a reduced number of users were benefited by them.

The advisability of having a standard object-oriented modeling language that facilitates the exchange of models among different developers and tools was recognized, and a design group was established in 1996 to propose such a language. The design group was composed of people who already have been involved in the design of other modeling languages or the programming of model libraries, and of people from industry. The proposed modeling language, which was named Modelica, incorporated ideas from existing modeling languages such as ALLAN, Dymola, NMF, ObjectMath, Omola, SIDOPS+ and Smile, and also introduced some new features.

Successive versions of the Modelica language have been released since 1997. An important milestone for the Modelica development happened in the year 2000: the foundation of the Modelica Association, a non-profit, non-governmental organization with the aim of developing and promoting Modelica for modeling, simulation and programming of physical and technical systems and processes. The website of the Modelica Association, `www.modelica.org`, hosts documentation about the language (specifications, scientific articles, tutorials, textbooks, etc.), and links to free model libraries and software. Among the free Modelica libraries, it is worth noting the Modelica Standard Library (MSL): a library that is developed and maintained directly by the Modelica Association.

This standardization effort, that ended up in the proposal of Modelica, also contributed to the popularization of the object-oriented modeling methodology, which is nowadays widely used in academia and industry. Many free and commercial model libraries written in the Modelica language are available. The easiness of reusing model components is one of the strong points of Modelica. A number of commercial and free software tools, that support the Modelica language or a part of it, are available. Some of these tools are CATIA, Dymola, JModelica.org, LMS Imagine.Lab AMESim, MapleSim, MathModelica, Modelicac, MWorks, SimulationX, OpenModelica, Scicos and Wolfram SystemModeler.

The research oriented to the improvement of the Modelica language, and the development of its software tools and model libraries, has aroused great interest over the last few years. As an example, the European projects EUROSYSLIB (Advanced Modelica Libraries), MODELISAR (Modelica-AUTOSAR Interoperability and Vehicle Functional Mock-up) and OPENPROD (Open Model-Driven Whole-Product Development and Simulation Environment) were awarded around 54 million euros during the 2009-12 period, and the work amount was 370 person-years. The MODRIO (Model Driven Physical Systems Operation) project, in which participate 38 industrial and academic partners, was awarded around 21 million euros during the 2012-16 period.

Aim and structure of the book

This book offers an introduction to the development and simulation of Modelica models for engineering applications. It has been written in the context of the Erasmus+ CBHE action "**InMotion - Innovative teaching and learning strategies in open modelling and simulation environment for student-centered engineering education**", Project No. "573751-EPP-1-2016-1-DE-EPPKA2-CBHE-JP", funded by the European Commission.

The target audience is bachelor's or master's level students, interested in modeling and simulation, and with a background in both physics and numerical methods.

The modeling methodology, the Modelica language features, and the use of modeling environments are explained through examples. This facilitates the use of this book in the context of **student-centered learning strategies**, such as problem-based learning, and project-based learning. In any case, readers are encouraged to install a Modelica modeling environment and simulate by themselves the models described in the book.

The book is structured into three parts: (i) continuous-time modeling; (ii) simulation of continuous-time models; and (iii) hybrid system modeling and simulation.

The modeling methodology and the Modelica features for continuous-time modeling are discussed in the **first part** of the book. The modeling methodology supported by Modelica, named object-oriented modeling, is discussed in Lesson 1. The mathematical formalism underlying the Modelica language and algorithms for simulating this type of model, named hybrid DAE system, are also discussed in the first lesson, and the use of Dymola and OpenModelica is introduced. The description of atomic models and model libraries in Modelica is discussed in Lessons 2 and 3.

The simulation of continuous-time Modelica models is addressed in the **second part** of the book. We have favored simplicity, clarity and readability over mathematical rigor. The main objective is to provide the reader with the minimum knowledge required for understanding the messages generated by the modeling environment (e.g., Dymola and OpenModelica) during the model translation and simulation. A broad range of topics are introduced: computational causality assignment, DAE index reduction, DAE initialization, state variable selection, and numerical methods for DAE systems. The analyses and symbolic manipulations that modeling environments perform on Modelica models are discussed in Lessons 4 and 5, and the numerical methods in Lesson 6.

The **third part** of the book is devoted to discuss hybrid modeling and simulation in Modelica. The formal specification of hybrid models, and the relationship of this specification with the simulation algorithm and the Modelica description, are described in Lesson 7. Numerical methods for event detection and handling are discussed in Lesson 8. Once again, simplicity has been favored over mathematical rigor. The objective is to provide the reader with the minimum knowledge required to understand the issues associated with the description of events, and variable structure models in Modelica. The goal is not to explain how to implement a simulator, but to explain how to design and implement models that can be simulated efficiently, without causing errors. Finally, the language features for describing time and state events, and runtime changes in the model mathematical description, are illustrated by a series of examples in Lesson 9.

Learning objectives

After studying the lessons and completing the proposed activities, students should be able to:

– Design model libraries applying the object-oriented modeling methodology.

– Develop and use model libraries in Modelica.

– Relate modeling hypotheses and numerical behavior of DAE hybrid models, including stiffness, algebraic loops, chattering, and high index.

– Formulate manually the simulation algorithm of small-dimension DAE-hybrid models, which includes assigning the computational causality, reducing the DAE index, applying numerical methods, and handling the events.

– Use Dymola and OpenModelica for editing, debugging and translating Modelica models, experimenting with the models and analyzing the simulation results.

About the authors

The authors are professors in the Departamento de Informática y Automática, at the Universidad Nacional de Educación a Distancia (UNED) in Madrid, Spain; and members of the research group on Modelling & Simulation in Control Engineering of UNED. Further information is available at: `www.euclides.dia.uned.es`

Part I

Continuous-time modeling

Lesson 1

Modeling methodology and tools

1.1 Introduction
1.2 Physical modeling paradigm
1.3 Object-oriented modeling
1.4 Modeling environments
1.5 Getting started with Modelica
1.6 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Relate the following concepts: physical modeling paradigm; equation-based modeling languages; computational causality of the model; non-causal modeling; causal modeling; object-oriented modeling methodology; object-oriented modeling language.

– Discuss features of the Modelica modeling language.

– Discuss the sequence of stages performed by the Modelica modeling environments for translating the Modelica model into executable code.

– Discuss the simulation algorithm for hybrid DAE systems implemented by the Modelica modeling environments.

– Use Dymola and OpenModelica for editing, debugging and translating Modelica models, experimenting with them, and analyzing the simulation results.

1.1 Introduction

Modeling and simulation of dynamical systems have many applications in Engineering, playing a fundamental role in system design, analysis, control and optimization. Support decision systems and training simulators are frequently based on mathematical modeling and computer simulation.

As simulation projects become larger and more complex, the impact of the modeling methodology and the software tools on the project cost is more evident. Adequate methodologies and tools are key success factors. Complex simulation projects typically require **working in teams**. Therefore, it is desirable that methodologies and tools facilitate splitting the modeling task among the team members, allowing them to work independently. Another key feature is **model reusability**.

The modeling methodology presented in this lesson, named **object-oriented modeling**, facilitates the model design, development, maintenance and reuse. The fundamentals of a modeling paradigm named **physical modeling** are also discussed, as it constitutes the conceptual foundation for the object-oriented modeling methodology and languages, including the Modelica language. The use of two of the most advanced and widely-used Modelica tools is introduced: Dymola and OpenModelica.

1.2 Physical modeling paradigm

Modelica is a modeling language designed to facilitate the application of the **physical modeling paradigm**. According to this paradigm, the model of a physical system is developed following three steps:

1. Analyze the system structure and divide the system into parts.
2. Analyze and describe the interaction among the parts.
3. Describe the internal behavior of each part, independently of the others, in terms of equations such as mass, energy and momentum balances, constitutive relationships, etc.

The model typically obtained by applying this paradigm is a **hybrid DAE model**, composed of algebraic equations, ordinary differential equations with derivatives with respect to time, and events. DAE stands for Differential-Algebraic Equation.

The languages conceived to support the physical modeling paradigm allow the model developer to describe the continuous-time part of the model using equations.

For this reason, these modeling languages are known as **equation-based languages**. Modelica belong to this type of modeling language.

An **equation** states an equality relationship between two expressions (i.e., `expression_1 = expression_2`) in which one or several model variables intervene. During the model simulation, the equation is employed to calculate the value of one of these variables.

In equation-based languages, the way in which the model developer writes an equation does not determine the variable to be calculated from the equation. Likewise, the order in which the model developer writes the model equations does not determine the evaluation order of the equations during the model simulation.

To illustrate this point, let's suppose that the constitutive relationship of an ideal resistor is described using an equation-based modeling language. The resistance has a constant, known value, R. The voltage drop (v) across the resistor pins and the electric current that flows through the resistor satisfy a linear relationship: the Ohm's Law. This equation can equivalently be written as:

$$\begin{array}{ccc} v = i \cdot R & i = v/R & R = v/i \\ v/i = R & v - i \cdot R = 0 & 0 = v - R \cdot i \end{array}$$

The **computational causality** of a model indicates which equation is used to evaluate each variable, or equivalently, which variable is evaluated from each equation. In equation-based modeling languages, the way in which the equations are written does not condition or inform about their computational causality. For this reason, equation-based modeling is also known as **non-causal modeling**.

In equation-based languages, the assignment of computational causality, and the symbolic manipulation and sorting of the model equations, are not carried out by the model developer. The modeling environment performs automatically these tasks. This feature greatly facilitates model development and reuse, reducing the time and effort required for completing simulation projects.

The computational causality of each equation does not only depend on itself, but also on the other model equations. For this reason, the computational causality of a model described by equations is a global property of the model.

Following up with the previous example, the computational causality of the resistor's constitutive relationship differs if the resistor is connected to a voltage or current generator. The models of the two circuits are shown below.

$$\begin{array}{l|l} [v] = f_v(time) & [i] = f_i(time) \\ v = [i] \cdot R & [v] = i \cdot R \end{array}$$

Each model is composed of two equations: the constitutive relationships of the generator and the resistor. The computational causality has been annotated writing within square brackets the variable that will be evaluated from each equation during the model simulation. Note that the equation that describes the resistor is the same in both models, but its computational causality is different.

In addition to using equations, Modelica allows to use **algorithms** for describing the model. An algorithm is a sorted sequence of assignments. An assignment has the following form: `variable := expression`.

The computational causality is explicitly indicated in assignments: the variable written on the left-hand side is the variable to calculate from the assignment, which is performed by evaluating the expression written on the right-hand side. The way of writing an algorithm indicates the assignment to be used for evaluating each variable and also the order in evaluating the variables. The modeling environment does not manipulate or sort the assignments of an algorithm. As the computational causality is explicitly indicated by the model developer, modeling using algorithms is referred to as **causal modeling**.

Modelica allows to encapsulate algorithms within **functions**, which can be invoked in any part of the model. A function has a well defined interface: output arguments, which are variables calculated from the algorithm, and input arguments, which are variables intervening in the right-hand expressions of the algorithm assignments.

Modelica also supports external functions. An **external function** is a Modelica function that encapsulates a call to a C, Fortran or Java function.

Modelica allows to define discrete-time variables and **events**. The trigger condition of an event is specified as a change in the value of a logical expression. Events can produce instantaneous changes in the value of the continuous-time and discrete-time state variables, and modify the mathematical description of the model.

Hybrid modeling in Modelica is based on the **synchronous data flow principle**. At any time, the active equations and algorithms describe relationships among variables that must be satisfied concurrently. The set of active equations can be composed of continuous-time equations (during the solution of the continuous-time problem), and by a combination of continuous-time and discrete-time equations (at the event execution). The evaluation order of the equations is automatically calculated by the modeling environment, so that it is unequivocally defined from which equation or algorithm has to be calculated each variable at every instant.

1.3 Object-oriented modeling

Object-oriented modeling is based on principles of modular and hierarchical modeling such as model reuse by composition; distinction between model interface and internal description; and information encapsulation. In addition, object-oriented modeling introduces new concepts such as model class; and model reuse by inheritance.

Modelica supports the object-oriented modeling methodology, facilitating model reuse both by composition and inheritance. Some of the language features are summarized below.

– **Modular and hierarchical modeling**. In the definition of the model classes, Modelica allows to distinguish between interface and internal description, and to encapsulate the information. The variables that don't belong to the interface can be declared as **protected variables**, so that they cannot be accessed from outside the class.

  To facilitate the component connection, Modelica allows to group the interface variables in **connectors**, and provides the syntax to define the connection between connectors.

  Likewise, Modelica establishes rules (inspired in the energy conservation principle) to relate the connector variables in the connection point. To this end, the connector variables are classified into the following two types:

  - **Across variables**, also known as **effort variables**, are those variables that are set equal at the connection point.
  - **Through variables**, also known as **flow variables**, are those variables whose sum is set equal to zero at the connection point.

– **Inheritance**. Modelica supports **multiple inheritance**, and the definition of **partial classes**, which define general properties of the class but cannot be instantiated. Partial classes facilitate the description of characteristics common to several classes.

– **Parametrization**. Modelica allows to modify the class **parameters** when the class is instantiated and inherited. The parameters can be time-independent variables. Also, the class of components or superclasses can be a parameter. This feature simplifies experimenting with models.

Table 1.1: Classes of the Modelica language.

Name	Usage
`type`	Definition of new variable types by extending the types predefined in the language.
`connector`	Definition of connectors (i.e., sets of interface variables) with the purpose of facilitating the description of component connections. Connectors cannot contain equations.
`model`	Definition of model classes.
`block`	Definition of model classes whose interface variables have the computational causality explicitly defined.
`record`	Definition of sets of variables and parameters. The purpose is facilitating the model parametrization. Records cannot contain equations.
`function`	Definition of functions that can encapsulate a Modelica algorithm, or a call to an external function written in C or Fortran.
`package`	Definition of model libraries. A package is a class that only can contain classes.

– **Annotations**. Modelica allows to include annotations in the class model, whose purpose is to define the graphical properties of the class icon and diagram, the model documentation, etc. The capability of incorporating this information in the model facilitates the model composition using a model graphical editor.

The definition of the model classes, packages, variable types, data records and functions is made using the **seven classes** provided by the Modelica language, which are shown in Table 1.1.

As discussed previously, the interaction among the components is described in terms of variables that are grouped in connectors, and classified in each connector into across and through, depending on whether they are equal at the connection point or their sum is equal to zero.

The connector variables can be selected in a way that component connections satisfy the conservation laws. For instance, suppose that the connector variables are selected as shown in Table 1.2, where the product of the across and through variable of each connector has power units. The across variables describe physical quantities that determine whether equilibrium exists, whereas the through variables describe physical quantities that restore the equilibrium. This selection guaranties

Table 1.2: Connector variables in different energy domains (*across* × *through* has power units).

Domain	**Across variable**	**Through variable**
Electrical	Voltage (V)	Current (A)
Mechanical translation	Velocity (m/s)	Force (N)
Mechanical rotation	Angular velocity (rad/s)	Torque (N·m)
Hydraulic	Pressure (N/m^2)	Volume flow rate (m^3/s)
Thermal	Temperature (K)	Entropy flow rate (W/K)
Chemical	Chemical potential (J/mol)	Molar flow rate (mol/s)

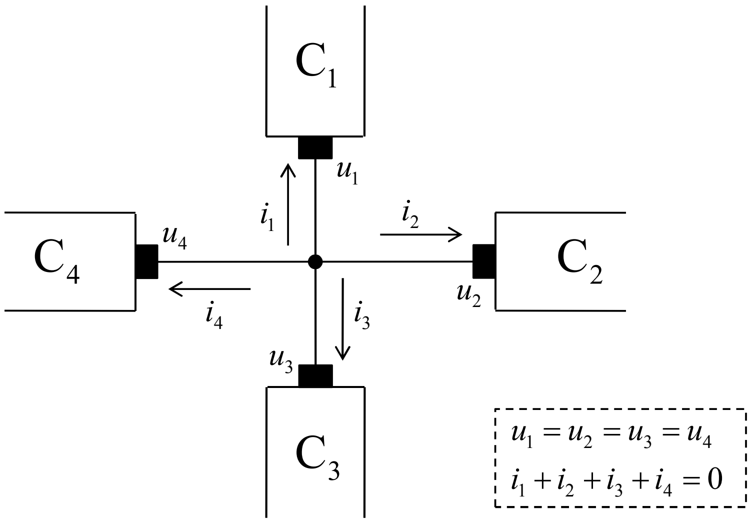


Figure 1.1: Connection among four components (C$_1$, C$_2$, C$_3$ and C$_4$). Connectors (filled rectangles) are composed of an across variable (u) and a through variable (i).

that if several connectors of the same domain are connected, then the **power is conserved** at the connection point.

Other selections that guarantee the energy conservation are possible. For instance, the heat flow rate is more frequently used than the entropy flow rate as through variable of the thermal domain. As the heat flow rate has power units, the energy conservation is described by the equation stating that the sum of the through variables at the connection point is zero.

Suppose that we want to model an electric circuit composed of current and voltage sources, resistors, capacitors, inductances, diodes, transistors, etc. The connection terminals of the electric components are named **pins**. The connection point of two or more pins is named circuit **node**. Let's consider a connection node of four components C$_1$, C$_2$, C$_3$ and C$_4$, as shown in Figure 1.1. Pins are represented in the figure as filled rectangles.

Selecting the electric connector variables as suggested in Table 1.2, the pin is modeled as a connector composed of an across variable and a through variable: the voltage (u) and current (i), respectively. Let's adopt the following **sign convention** for through variables: the through variable is defined as entering the connector. In other words, positive current enters the component. The connection among the four components is translated by the modeling environment into four equations: three equations stating the equality of the voltages ($u_1 = u_2 = u_3 = u_4$) and one equations stating the current conservation ($i_1 + i_2 + i_3 + i_4 = 0$).

1.4 Modeling environments

A modeling environment is a computer program aimed to edit, check, debug, translate into executable code, and simulate models described using a certain modeling language. Modeling environments typically include a graphic model editor, that allows to compose models from model libraries by dragging, dropping and connecting components, facilitating setting parameter values, accessing the code and documentation of the components, and navigating through the model hierarchy. Modeling environments also facilitate plotting the simulation results and exporting them to file.

The translation of the Modelica model into executable code is performed through a sequence of stages. The first set of operations are intended to translate the Modelica model into a flat model. The operations of this **translation stage** include performing lexical, syntactic and semantic analyses, type checking, undoing inheritance and composition, and generating connection equations.

The **flat model** is equivalent to the object-oriented model, but with the hierarchy, composition and inheritance undone. A flat model is composed by declarations of constants, parameters and variables, and equations and algorithms. The flat-model variables are named using dot notation, in accordance with the hierarchical structure of the Modelica model.

Connection among components is described in Modelica using **connect sentences**. These sentences are replaced by equations in the translation stage. At each connection node, the across variables are set equal and the sum of the through variables is set to zero.

Equations that state equality between two variables are named **trivial equations**, and the variables are named **alias variables**. The across variables of the connected connectors are alias variables. The equations that relate them are trivial equations, which are added to the model as a result of the connect-sentence translation.

The set of operations that is performed next on the flat model constitutes the **analysis and optimization stage**. An obvious optimization consists in substituting the alias variables, removing the trivial equations from the simulation model.

Equations don't contain explicit information on their computational causality, which is calculated by the modeling environment from analyzing the complete model. As a result of this analysis, named **assignment of the computational causality**, the evaluation order of the equations and algorithms is obtained, and also which variable has to be evaluated from each equation.

In addition to the assignment of the computational causality, other analyses and manipulations performed on the model are reduction of the DAE index, symbolic manipulation of linear equations and systems of simultaneous equations, tearing of non-linear systems of simultaneous equations, and optimization of expression evaluations.

The model obtained of performing these operations is named the **sorted and solved model**. Its simulation algorithm (typically programmed in C language) is automatically generated by the modeling environment, and compiled and linked with libraries of numerical methods, to obtain the simulation executable file.

As Modelica models are hybrid, the simulation algorithm implemented by the Modelica modeling environments combines the solution of the continuous-time part of the model, with the detection and handling of the events. The algorithm basically works as follows:

1. The continuous-time part of the model is solved using **numerical integration** methods for DAE systems. As discrete-time variables only change their values at event instants, these variables are assumed to be constant during the integration of the continuous-time part of the model. The model is evaluated at time instants determined by the step size of the integration algorithm. After each evaluation, the event conditions are checked.

2. If an event condition is satisfied, it means that the associated event has been triggered within the last time step of the integration algorithm. Then, the integration is halted and an iterative method to accurately **locate the event trigger time** is started. A small interval that includes the trigger time is found. The interval length is smaller than a predefined value that depends on

the precision of the event-location method. It is assumed that the trigger time is the right endpoint of the calculated interval.

3. Once the event trigger time has been calculated, the integration method is used to evaluate the model at the event instant, prior to executing the event actions. Next, the event actions are executed: the continuous-time part of the model and the active discrete-time equations are solved simultaneously. This is known as solving the **restart problem**. If several events are triggered simultaneously, the discrete-time equations of all these triggered events are considered in the restart problem. Observe that two model evaluations are performed at the event time, one prior to the event execution and the other when executing the event (for solving the restart problem), resulting in two values of the model state: the **previous** value and **new** value, respectively.

4. Once the restart problem has been solved, the event conditions are checked again. If one or several events are triggered, then the new restart problem, which is composed of the continuous-time part of the model and the discrete-time equations of the triggered events, is posed, solved, and the event conditions are checked again. This process, named execution of an **event chain**, continues until no more events are triggered. Then, the numerical integration of the continuous-time part of the model is resumed, using as initial condition the values calculated from solving the last restart problem.

The state of hybrid models evolves by continuous change over time of the continuous-time state variables, and by instantaneous changes in the total state, described by the continuous-time and discrete-time state variables, known as **events**. The description of an event has two parts:

1. The **trigger condition**, also known as **activation condition** or simply event **condition**. Depending on the activation condition, events are classified into state and time events.

   – **State events** are those whose condition depends on any continuous-time variable.

   – **Time events** are those whose condition does not depend on continuous-time variables. The condition of time events may depend on time and discrete-time variables.

2. The **action** to perform. It typically consists in producing an instantaneous change in the total state of the model or the equations that describe the model

behavior. The model is said to have a **variable structure** if its mathematical description may change during the simulation.

As time-event conditions don't depend on continuous-time variables, the trigger instant of time events can be scheduled in advance during the simulation. The handling of this type of events is typically performed using an **event calendar**, where future events are stored in order. When the first scheduled event is within the next time-step of the integration algorithm, the length of the time step is reduced, so that the model is evaluated precisely at the event instant.

In general, state events cannot be scheduled in advance and for this reason are detected by checking their event conditions during the solution of the continuous-time problem. The event condition of each state event is expressed as a logical expression, so that the event condition is fulfilled when the value associated to the logical expression changes.

Dymola and OpenModelica are two widely-used Modelica modeling environments that constitute the state of the art. Dymola is commercial software. However, there is a free demonstration version, limited to models up to 10 state variables, that allows to simulate most of the models described in this text. OpenModelica is free software. The use of Dymola and OpenModelica is introduced in the next section.

1.5 Getting started with Modelica

A simple, plane pendulum is modeled by the following two equations

$$\frac{d\varphi}{dt} = w \tag{1.1}$$

$$L \cdot \frac{dw}{dt} = -g \cdot \sin(\varphi) \tag{1.2}$$

were φ represents the pendulum angle with respect to the vertical, w the angular velocity, L the pendulum length, and g the gravitational acceleration.

This model is described in Modelica as shown in Modelica Code 1.1. The Modelica description of the model has two sections, containing the variable declaration and the equations respectively. The variables are declared first. The **equation** keyword signals the beginning of the equation section.

The model has two continuous-time variables of real type that represent the angle (`phi`) and the angular velocity (`w`); a parameter of real type that represents

```
model pendulum
   constant Real g =  9.81 "Gravitational acceleration";
   parameter Real L = 1 "Length of the rigid rod";
   Real phi(start=0.1, fixed=true) "Angle";
   Real w(start=0, fixed=true) "Angular velocity";
equation
   der(phi) = w;
   // Equation of motion
   L*der(w) = -g*sin(phi);
end pendulum;
```

Modelica Code 1.1: Model of a simple plane pendulum.

the pendulum length (`L`); and a constant of real type that represents the gravitational acceleration (`g`).

The values of constants and parameters don't change during the simulation. The **parameter** values may be changed between simulation runs. The setting of parameter values is a part of the simulation experiment definition. On the contrary, the **constant** values cannot be changed at the experiment definition. In this example, the parameter and constant values are set in the variable declaration: `L = 1`, `g = 9.81`. The value of `L` may be modified between simulation runs.

The continuous-time variables `phi` and `w` appear differentiated in the model. Both are selected by-default as continuous-time **state variables**. In that way, their time derivatives are calculated from the model equations, and `phi` and `w` are calculated by numerical integration of their time derivatives.

The **initial values** of `phi` and `w` are specified by setting the value of the attribute **start** to the variable initial value, and setting the **fixed** attribute to true. The initial value of `phi` is 0.1 radians and the initial value of the angular velocity is zero.

It is possible to include **comments** in the Modelica code. Comments associated to a declaration are written within double quotation marks `"`, before the semicolon that signals the end of the sentence. These comments are typically shown by the model editor, accompanying the variable name, as an aid for understanding the variable meaning.

Modelica allows also to include comments intended only for model developers. The text written in a line after two slashes `//` will be ignored by the model editor. It is also possible to include multi-line comments: the text written between the symbols `/*` and `*/` will be ignored.

The model equations are written after the **equation keyword**. Observe that the **time derivatives** of the `phi` and `w` variables are `der(phi)` and `der(w)`, respectively. The way of writing the equations and their order are irrelevant. The modeling environment will analyze the model to find out which variable has to be evaluated from each equation and in which order these evaluations have to be made, and will manipulate symbolically the equations.

1.5.1 Dymola

Firstly, let's edit and simulate this model using Dymola. The graphic user interface (GUI) of Dymola version 2017 is shown in Figure 1.2. The modeling view of the GUI is displayed on launch. The main window is divided into three windows: the upper left window (Packages) allows to navigate through the model libraries, the lower left window (Component Browser) shows the components of the model that is being edited, and the right window allows to edit the model. The library named Modelica shown in the Packages window is the **Modelica Standard Library** (MSL). It is opened by-default when Dymola is launched.

The menu and button bars located at the top of the main window allow to open files containing Modelica models (text files with .mo extension), save models to file, and model editing (defining graphic attributes, connecting components, etc.) and checking. The two buttons at the lower-right part of the main window, labeled as *Modeling* and *Simulation* respectively, allow to switch between the **modeling view** (displayed by-default on launch), which allows to load, save, edit and check models, and the **simulation view**, which allows to translate and simulate the model, define the experiment, and plot the results.

The pendulum model can be edited and simulated following these steps:

1. **Create a new model**. Choose *File* > *New* > *Model* in the menu bar, and write the name of the new model: *pendulum*. Dymola asks whether the new model has to be included within any of the libraries shown in the Packages window. As our answer is negative in this case, the new model is created in the upper hierarchical level of the library tree displayed in the Packages window.

2. **Write the Modelica code of the model**. Choosing *Window* > *View* > *Modelica Text* or pressing the *Modelica Text* button, the right window changes to the code editing mode. Then, we can write the Modelica description of the model (see Figure 1.3). Choosing *File* > *Save* saves the model to file. Modelica classes are written in text files with *.mo* extension. The directory where the file is saved is selected as the working directory for the session. It can be changed choosing *File* > *Change directory.*

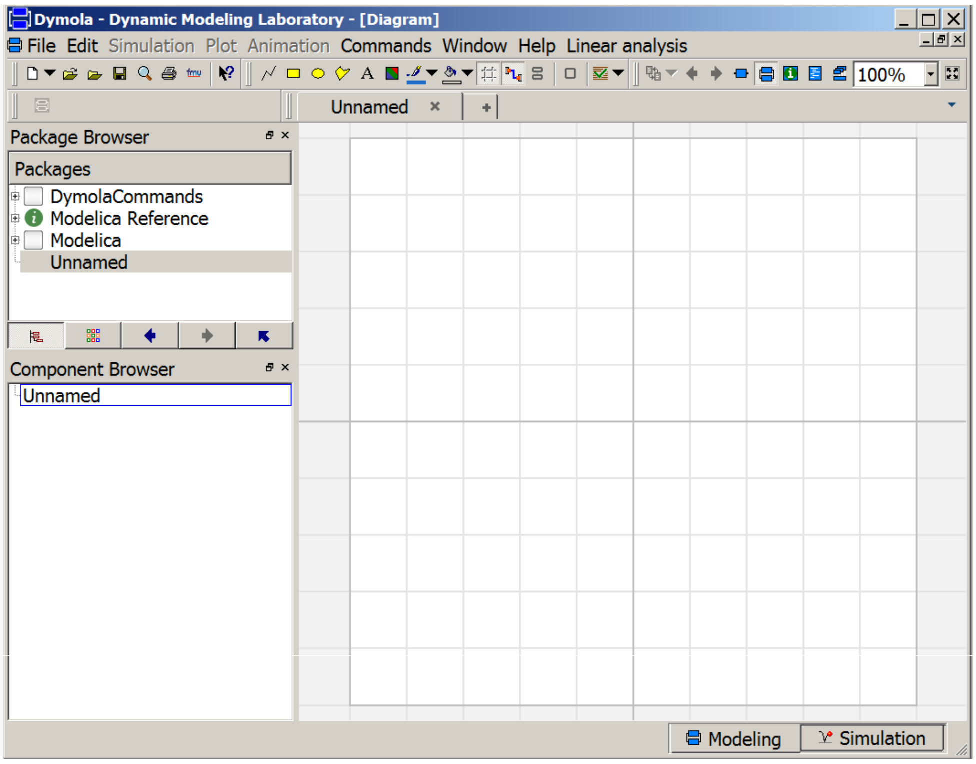
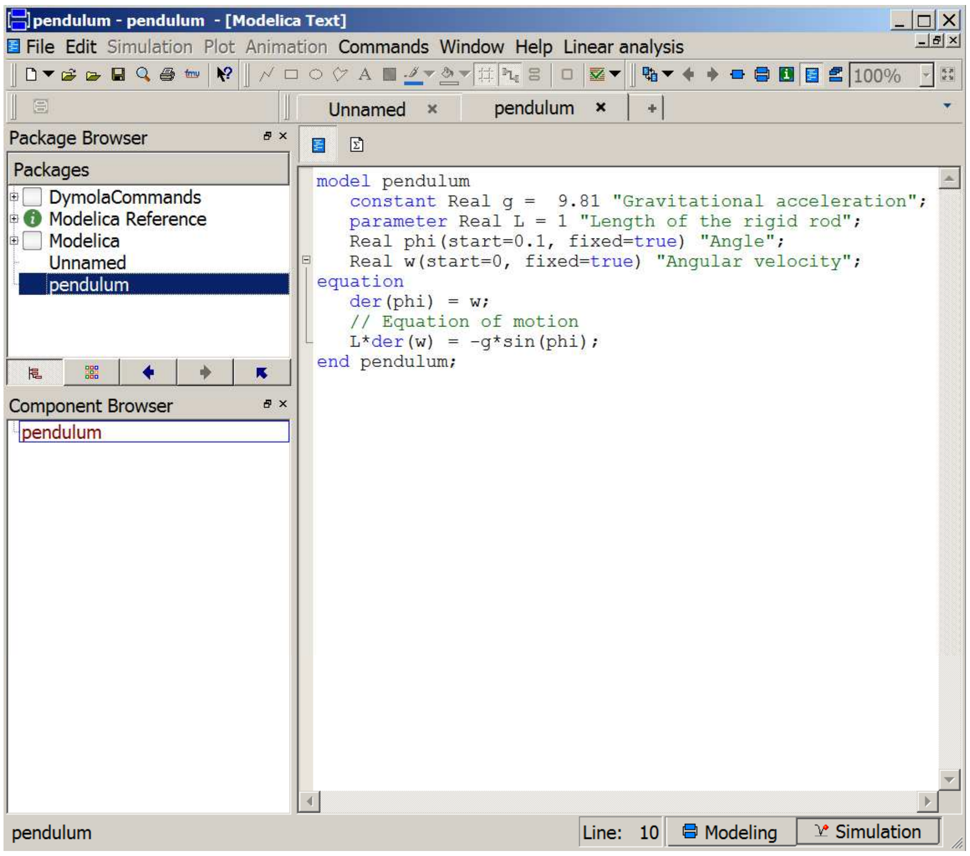
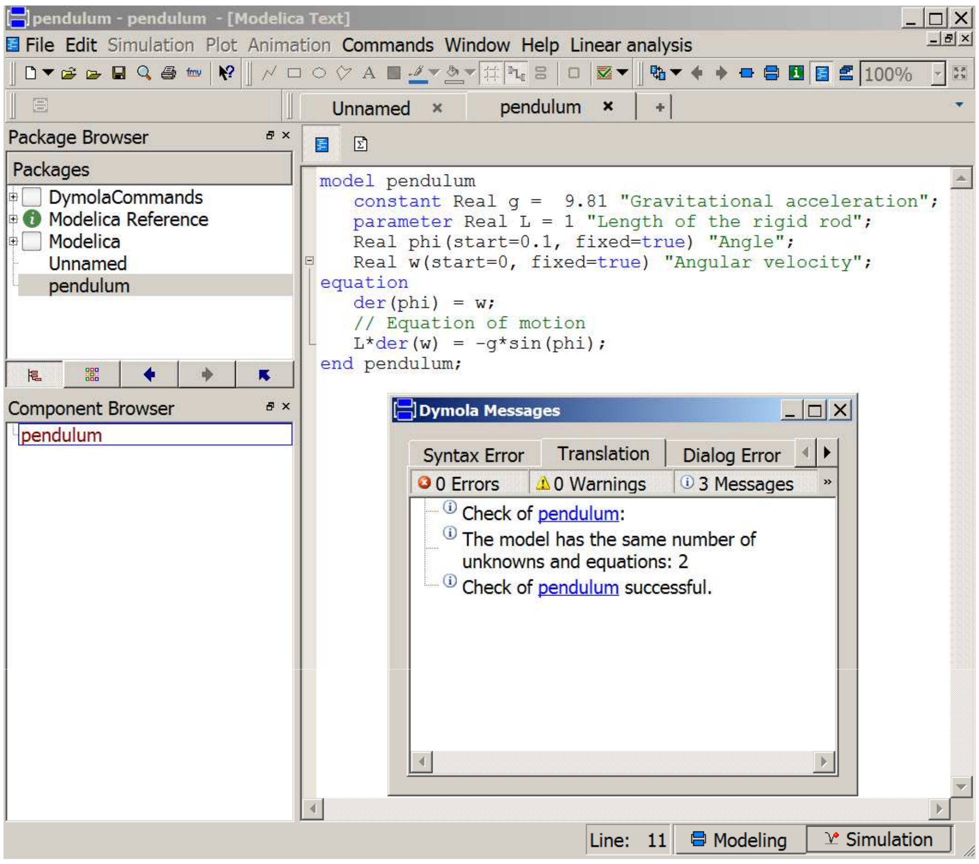


Figure 1.2: Model edition window of Dymola version 2017.



```
model pendulum
   constant Real g =  9.81 "Gravitational acceleration";
   parameter Real L = 1 "Length of the rigid rod";
   Real phi(start=0.1, fixed=true) "Angle";
   Real w(start=0, fixed=true) "Angular velocity";
equation
   der(phi) = w;
   // Equation of motion
   L*der(w) = -g*sin(phi);
end pendulum;
```

Figure 1.3: Edition of the pendulum model.



```
model pendulum
   constant Real g =  9.81 "Gravitational acceleration";
   parameter Real L = 1 "Length of the rigid rod";
   Real phi(start=0.1, fixed=true) "Angle";
   Real w(start=0, fixed=true) "Angular velocity";
equation
   der(phi) = w;
   // Equation of motion
   L*der(w) = -g*sin(phi);
end pendulum;
```

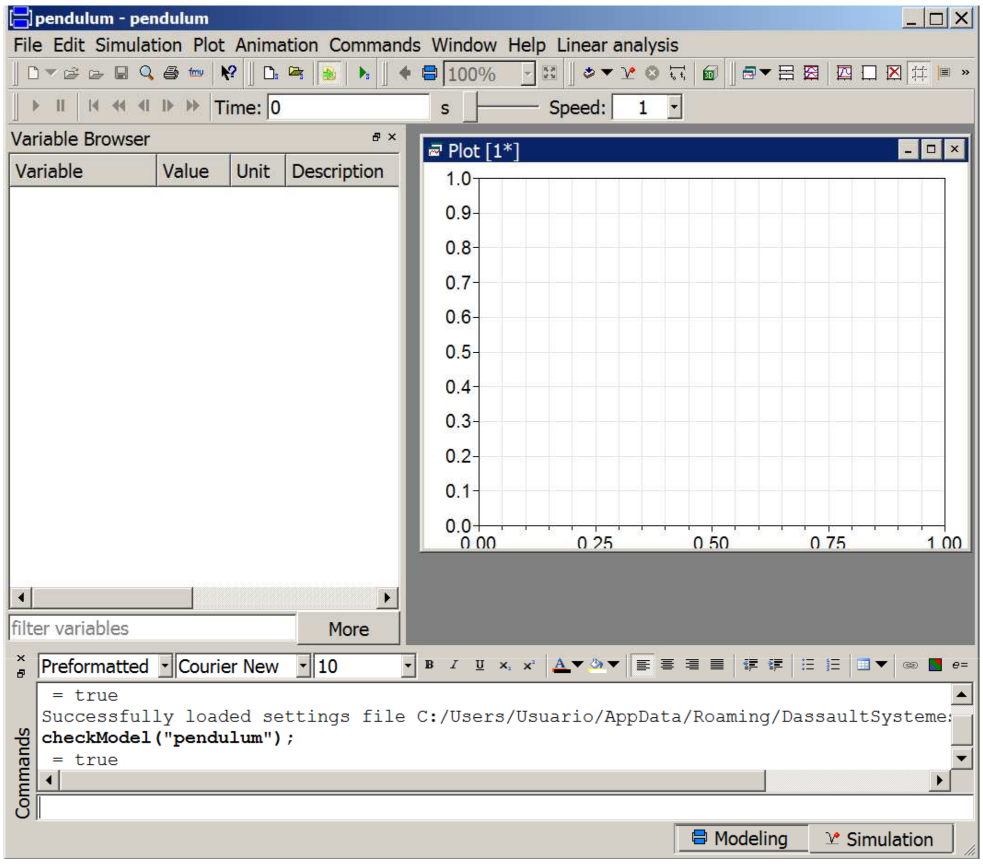
0 Errors 0 Warnings 3 Messages

Check of pendulum:
The model has the same number of unknowns and equations: 2
Check of pendulum successful.

Figure 1.4: Checking of the pendulum model.

3. **Check the model**. Choosing *Edit* > *Check* > *Normal* or pressing the *Check* button, Dymola checks that the model being edited does not contain syntax errors. Also, Dymola displays the number of unknown variables and the number of equations (see Figure 1.4). Both numbers have to be equal for the model to be simulated.

4. **Translate the model**. Clicking the *Simulation* button, located at the lower part of the main window, Dymola changes to the **simulation view** (see Figure 1.5). The name of the model to be translated and simulated is shown in the upper border of the main window: pendulum. The translation of the model is performed selecting *Simulation* > *Translate* > *Normal* or pressing the *Translate* button. Dymola translates the Modelica model into C language. The generated C-file, named *dsmodel.c*, is compiled and an executable file named *dymosim.exe* is created. Both files are saved in the working directory. The Dymola Messages window displays information on the model, including the number of unknown variables, equations, number and size of the systems of simultaneous equations, and also lists the variables that have been selected as continuous-time state variables (see Figure 1.6).

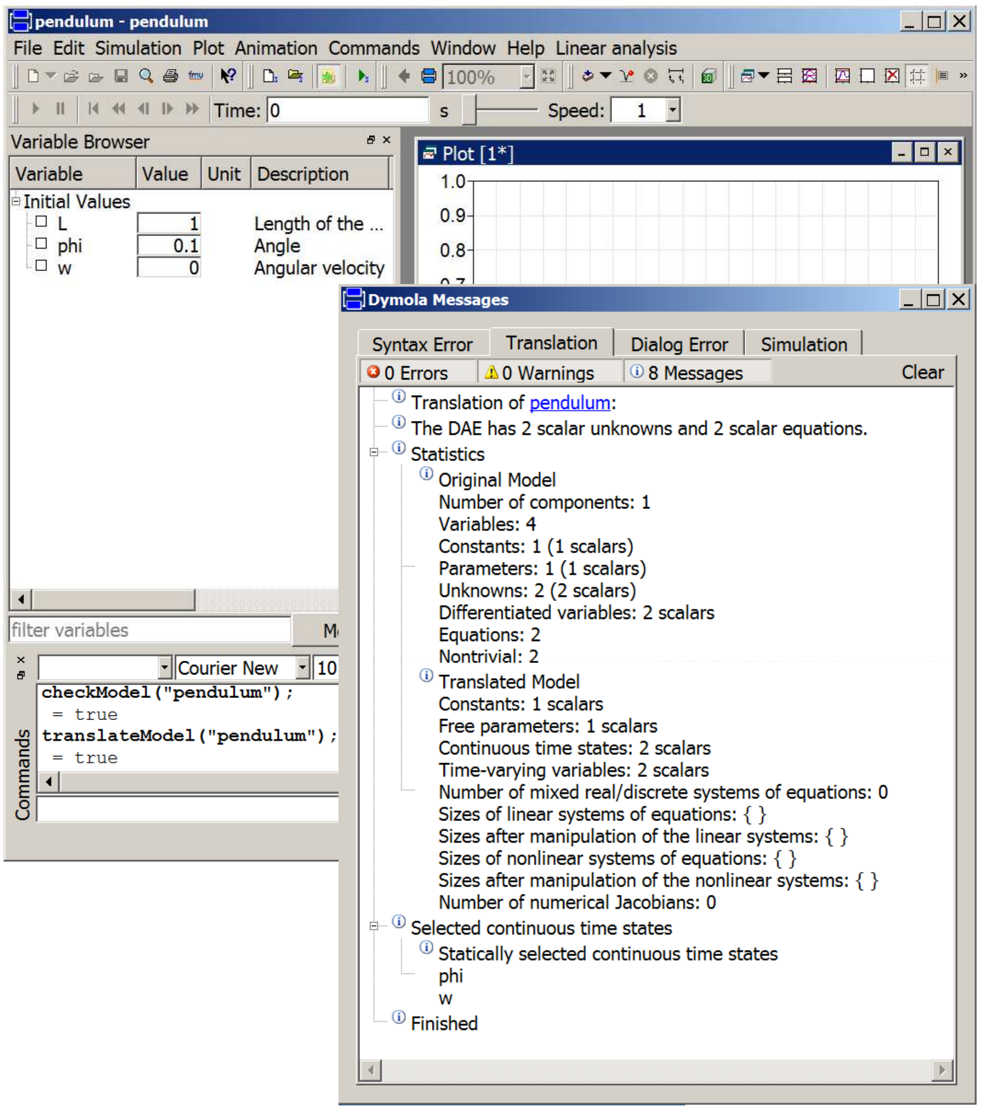


```
 = true
Successfully loaded settings file C:/Users/Usuario/AppData/Roaming/DassaultSysteme:
checkModel("pendulum");
 = true
```

Figure 1.5: Simulation window.

5. **Initialize the model**. After translating the model, the *Variable Browser* window (see Figure 1.6) shows the variables whose value can be set to initialize the model. In the pendulum model, these are the angle and the angular velocity (state variables), and the length (parameter). The values shown are the values assigned to these quantities in the model, being now possible to change these values. From the value of these quantities and the model equations, Dymola will be able to solve the model at the initial time of the simulation. The parameter value will be kept constant during all the simulation. The derivatives of the angle and the angular velocity will be calculated solving the model equations, and the angular velocity and the angle will be calculated by numerical integration of their derivatives.

6. **Define the experiment**. Choosing *Simulation* > *Setup* or clicking on the *Setup* button, the *Simulation setup* window opens. The initial and final simulation time, the length or number of output intervals, and the integration method and its tolerance or step size, are specified in the *General* tab. In this example, the final time is set to 5 (see Figure 1.7).

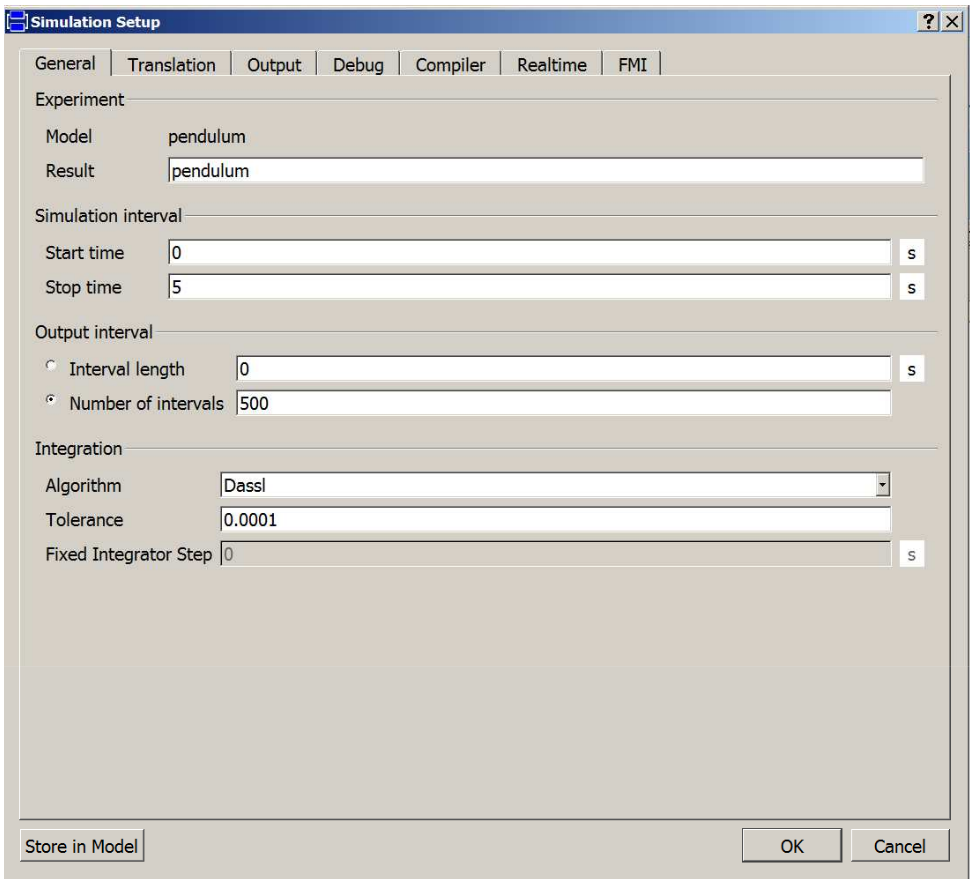


Variable	Value	Unit	Description
Initial Values			
L	1		Length of the ...
phi	0.1		Angle
w	0		Angular velocity

```
checkModel("pendulum");
 = true
translateModel("pendulum");
 = true
```

```
Translation of pendulum:
The DAE has 2 scalar unknowns and 2 scalar equations.
Statistics
  Original Model
  Number of components: 1
  Variables: 4
  Constants: 1 (1 scalars)
  Parameters: 1 (1 scalars)
  Unknowns: 2 (2 scalars)
  Differentiated variables: 2 scalars
  Equations: 2
  Nontrivial: 2
  Translated Model
  Constants: 1 scalars
  Free parameters: 1 scalars
  Continuous time states: 2 scalars
  Time-varying variables: 2 scalars
  Number of mixed real/discrete systems of equations: 0
  Sizes of linear systems of equations: { }
  Sizes after manipulation of the linear systems: { }
  Sizes of nonlinear systems of equations: { }
  Sizes after manipulation of the nonlinear systems: { }
  Number of numerical Jacobians: 0
Selected continuous time states
  Statically selected continuous time states
  phi
  w
Finished
```

Figure 1.6: Model translation.


Simulation Setup

Experiment

Model pendulum

Result pendulum

Simulation interval

Start time 0 s

Stop time 5 s

Output interval

Interval length 0 s

[✓ Number of intervals] 500

Integration

Algorithm Dassl

Tolerance 0.0001

Fixed Integrator Step 0 s

Figure 1.7: Experiment setup.

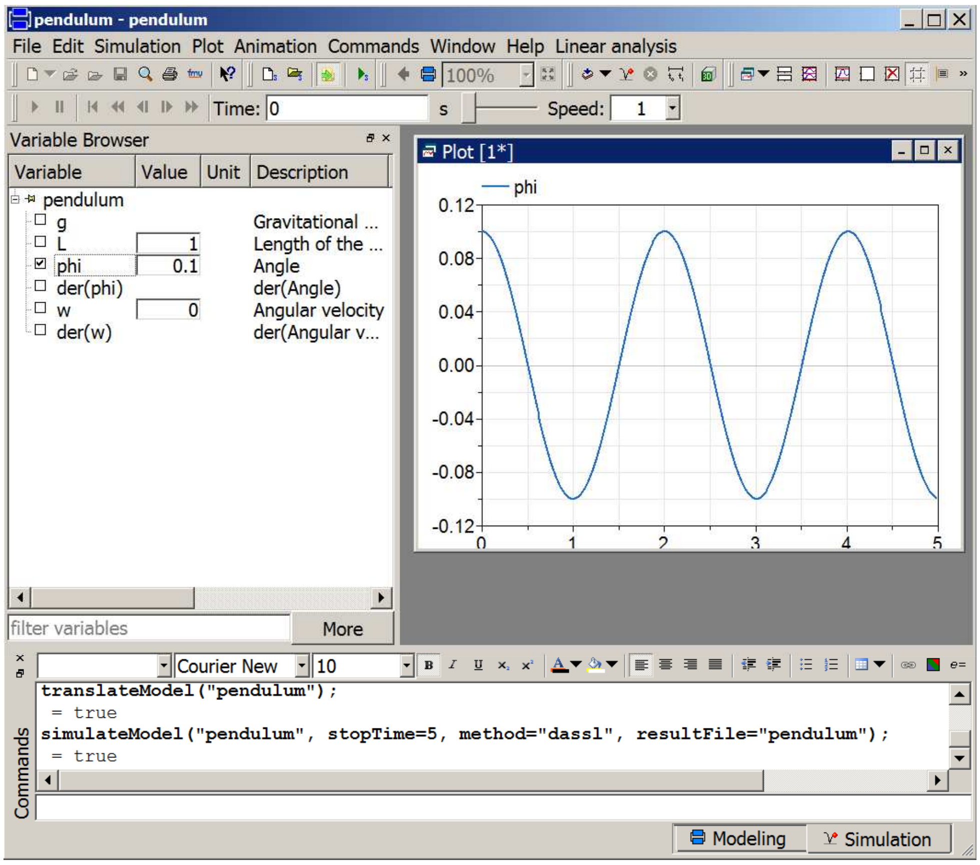
The *Translation* tab allows to specify additional information to be generated by Dymola on the translation, for instance, the flat model, or the sorted and solved model.

The *Output* tab allows to configure the type and amount of data to store during the simulation. Typically, the value of all variables not declared as reserved is saved at equispaced communication instants, and at events.

The *Debug* tab allows to specify additional information on the model numerical solution to be reported. The *Realtime* tab allows to synchronize the simulated time with real time.

The *Store in Model* button, located at the bottom of the *Simulation Setup* window, writes in the model an annotation describing the actual experimental configuration.

The experiment can also be described using the Modelica script language: the commands are written in a text file with *.mos* extension that is opened and executed selecting *Simulation* > *Open Script*.



```
translateModel("pendulum");
 = true
simulateModel("pendulum", stopTime=5, method="dassl", resultFile="pendulum");
 = true
```

Figure 1.8: Model simulation.

7. **Run the simulation and plot the results**. The simulation run starts choosing *Simulation* > *Simulate* or pressing the *Simulate* button (see Figure 1.8). Dymola automatically generates a text file named *dsin.txt* with the experiment description, including the information to initialize the model and configure the numerical methods, a list of all the model variables, and the initial and final time. The experiment description file can be also obtained by executing the following command from a Windows shell: `dymosim -i`

   Dymola launches the simulation by executing *dymosim.exe*, which reads the experiment file (*dsin.txt* by-default). The final state of the model is stored in a file named *dsfinal.txt*, which has the same form as the file describing the experiment.

   The results can be plotted during the simulation run and once the simulation is finished. The variable names are displayed in the *Variable Browser* window. Click on a variable to visualize it in the Plot window. By default, the horizontal axis represents time, but any other variable can be selected as independent variable. The simulation results are saved to a Matlab format file, whose name is the model name with *.mat* extension. If the option *Textual data format* is selected in the *Output* tab of the *Simulation Setup* window (see again Figure 1.7), then the simulation results are also written to a text file. Dymola writes these files in the working directory.

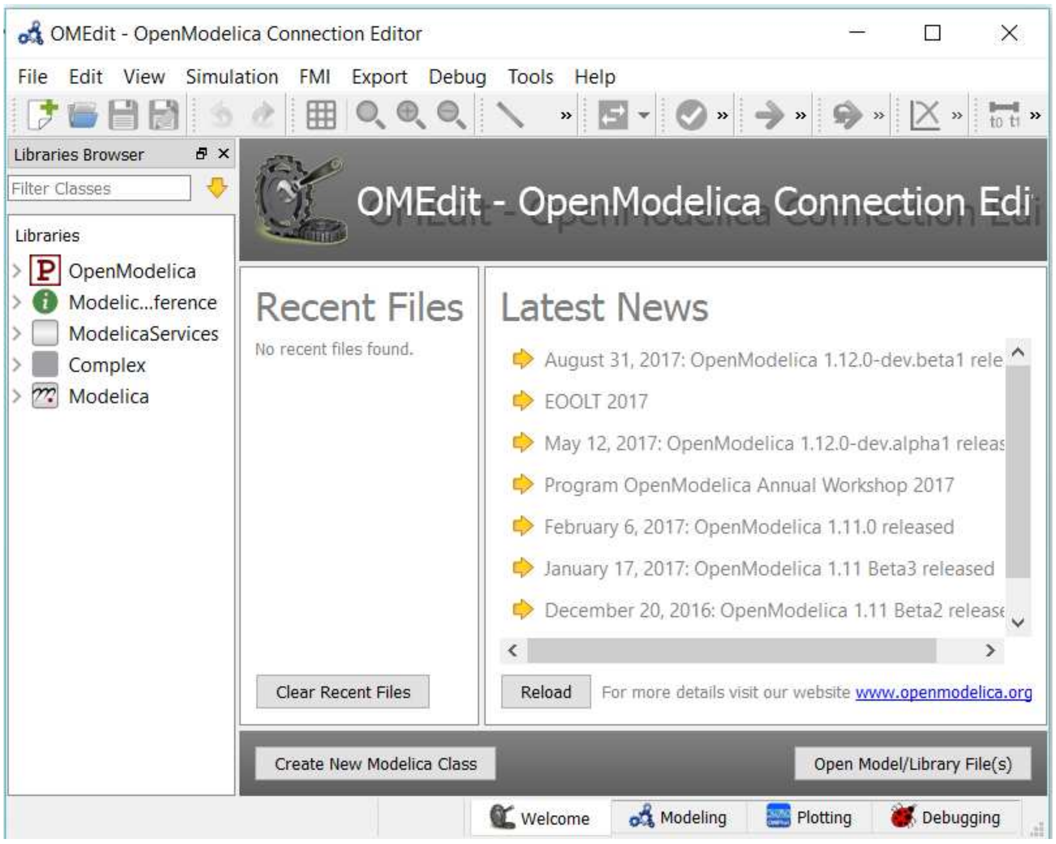
1.5.2 OpenModelica

Some guidelines to use the modeling and simulation environment OpenModelica are provided below. To this end, the steps required to edit, check and simulate a model using OpenModelica are described. We use the same example employed to describe Dymola: the pendulum shown in Modelica Code 1.1.

The OpenModelica environment has been developed by the Open Source Modelica Consortium (OSMC), it can be freely downloaded and there exist versions for Windows, Linux and Mac Operating Systems. The core of this environment is the Modelica Compiler, named OpenModelica Compiler, that transforms the Modelica code into C code. This environment includes many tools to interface between the compiler and the user. These tools are the OpenModelica Connection Editor (OMEdit), the Interactive OpenModelica Shell (OMShell), OpenModelica Notebook (OMNotebook), DrControl, OpenModelica Equation Model Debugger, OMOptim, Modelica Development Tooling (MDT), OpenModelica Python Interface (OMPython). We are going to use OMEdit as the user interface.

After installing OpenModelica in a Windows computer, OMEdit can be opened by simply executing *OMEdit.exe*, located in the *bin* directory of the OpenModelica installation. After executing it, the window shown in Figure 1.9 is displayed. This windows has the following three tabs: “Welcome”, “Modeling”, “Plotting” and “Debugging”. The “Welcome” tab is divided into three areas: an area to navigate in the model libraries (*Libraries Browser*), an area showing the Modelica files that have been opened recently (*Recent Files*) and an area with the last news about OpenModelica (*Latest News*). There are buttons in the top of the window, which allow the user to open new models stored in files, save models to a file, define graphical attributes, and perform operations in the models such as the syntactic checking of models, simulation, etc. The “Modeling” tab allows to edit models and the “Plotting” tab allows to visualize the simulation results.

The steps required to edit, check and simulate the pendulum example are briefly described below.



No recent files found.

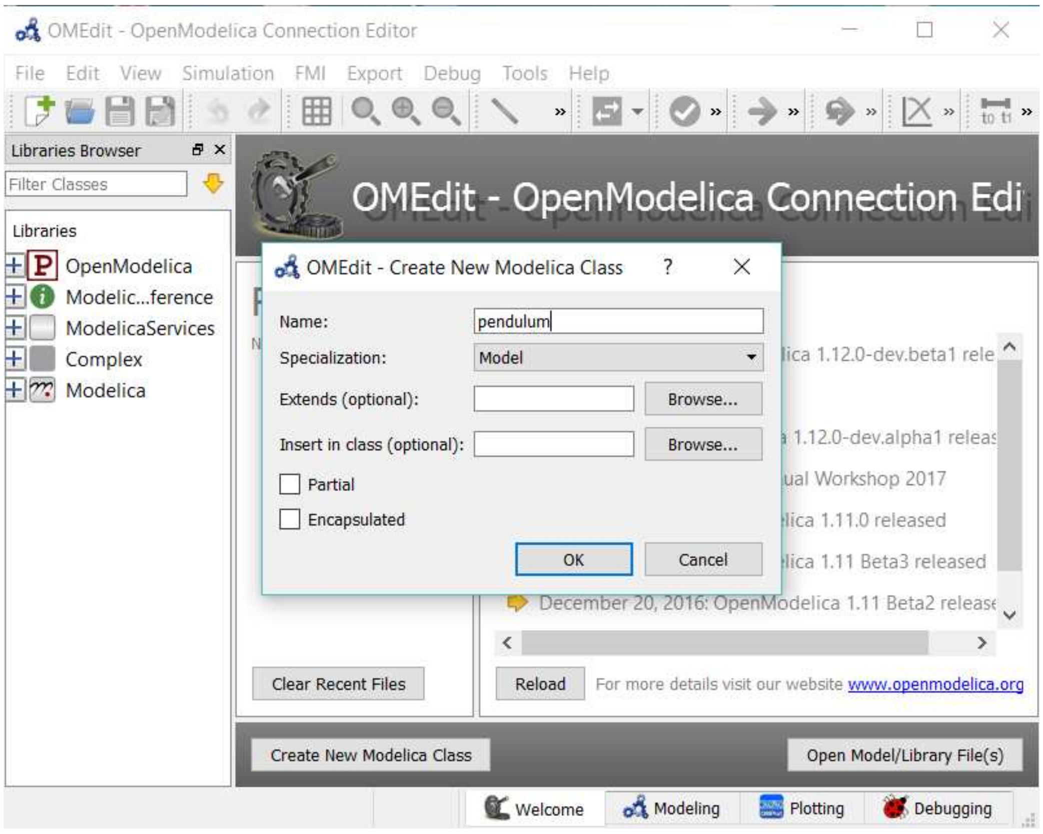
- August 31, 2017: OpenModelica 1.12.0-dev.beta1 rele
- EOOLT 2017
- May 12, 2017: OpenModelica 1.12.0-dev.alpha1 releas
- Program OpenModelica Annual Workshop 2017
- February 6, 2017: OpenModelica 1.11.0 released
- January 17, 2017: OpenModelica 1.11 Beta3 released
- December 20, 2016: OpenModelica 1.11 Beta2 release

For more details visit our website www.openmodelica.org

Figure 1.9: Window obtained after executing OMEdit.exe.

1. **Create a new model**. Choose *File > New Modelica class* in the menu bar. Then, a window is displayed (see Figure 1.10). The user can set the name of the new class in the *Name* box, and the type of the Modelica class in the *Specialization* box. By clicking on the *Partial* box, we are specifying that this new class is partial. This means that this new class can be inherited but not instantiated . If the new model inherits from other models, we can specify the name of the superclasses in the *Extends* box. Additionally, if the new model is part of an existing library, in the *Insert in class* box we can specify where it has to be included. In this example, we only fill in the first box with the name of the model.

2. **Write the Modelica code of the model**. By clicking on the button *Text View* (see Figure 1.11), a window is displayed where the Modelica code of the new *pendulum* model can be written. In this window we can choose to show the model icon, diagram or documentation by clicking the *Icon View*, *Diagram View* or *Documentation View* buttons, respectively. By clicking on the *Check Model* button, OpenModelica checks that the model has no errors, and shows a dialog window with information about the number of variables and equations.



OMEdit - Create New Modelica Class

Name: pendulum

Specialization: Model

Extends (optional):

Insert in class (optional):

☐ Partial

☐ Encapsulated

ica 1.12.0-dev.beta1 rele

1.12.0-dev.alpha1 releas

ual Workshop 2017

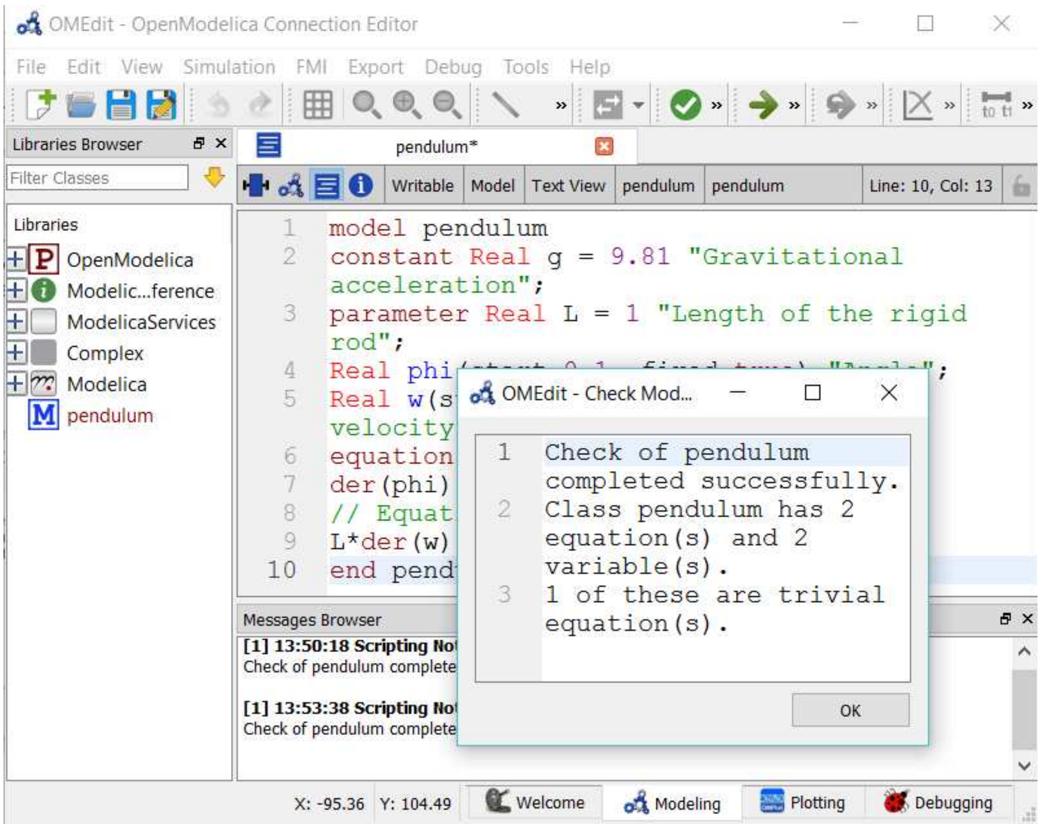
ica 1.11.0 released

ica 1.11 Beta3 released

December 20, 2016: OpenModelica 1.11 Beta2 release

For more details visit our website www.openmodelica.org

Figure 1.10: Setting a new Modelica class in OMEdit.



```
model pendulum
constant Real g = 9.81 "Gravitational
acceleration";
parameter Real L = 1 "Length of the rigid
rod";
Real phi                              ";
Real w(s
velocity
equation
der(phi)
// Equat
L*der(w)
end pend
```

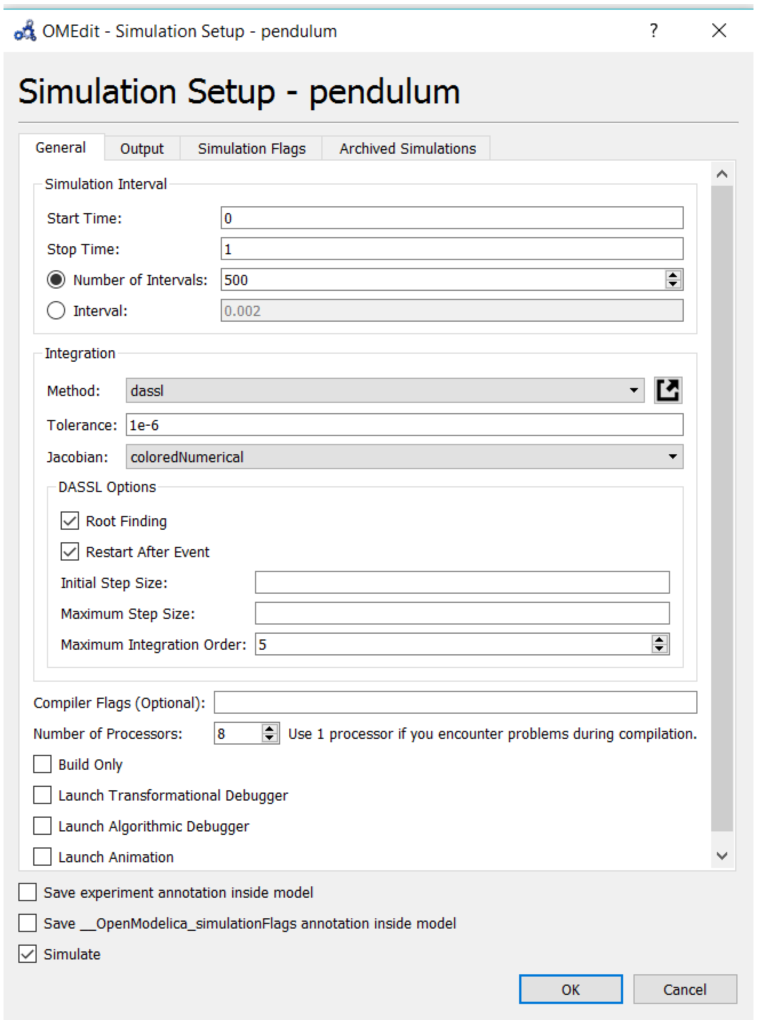
OMEdit - Check Mod...

```
Check of pendulum
completed successfully.
Class pendulum has 2
equation(s) and 2
variable(s).
1 of these are trivial
equation(s).
```

[1] 13:50:18 Scripting No
Check of pendulum complete

[1] 13:53:38 Scripting No
Check of pendulum complete

Figure 1.11: Editing a new model in OMEdit.



OMEdit - Simulation Setup - pendulum

Simulation Setup - pendulum

General | Output | Simulation Flags | Archived Simulations

Simulation Interval

Start Time: 0

Stop Time: 1

(●) Number of Intervals: 500

( ) Interval: 0.002

Integration

Method: dassl

Tolerance: 1e-6

Jacobian: coloredNumerical

DASSL Options

☑ Root Finding

☑ Restart After Event

Initial Step Size:

Maximum Step Size:

Maximum Integration Order: 5

Compiler Flags (Optional):

Number of Processors: 8 Use 1 processor if you encounter problems during compilation.

☐ Build Only

☐ Launch Transformational Debugger

☐ Launch Algorithmic Debugger

☐ Launch Animation

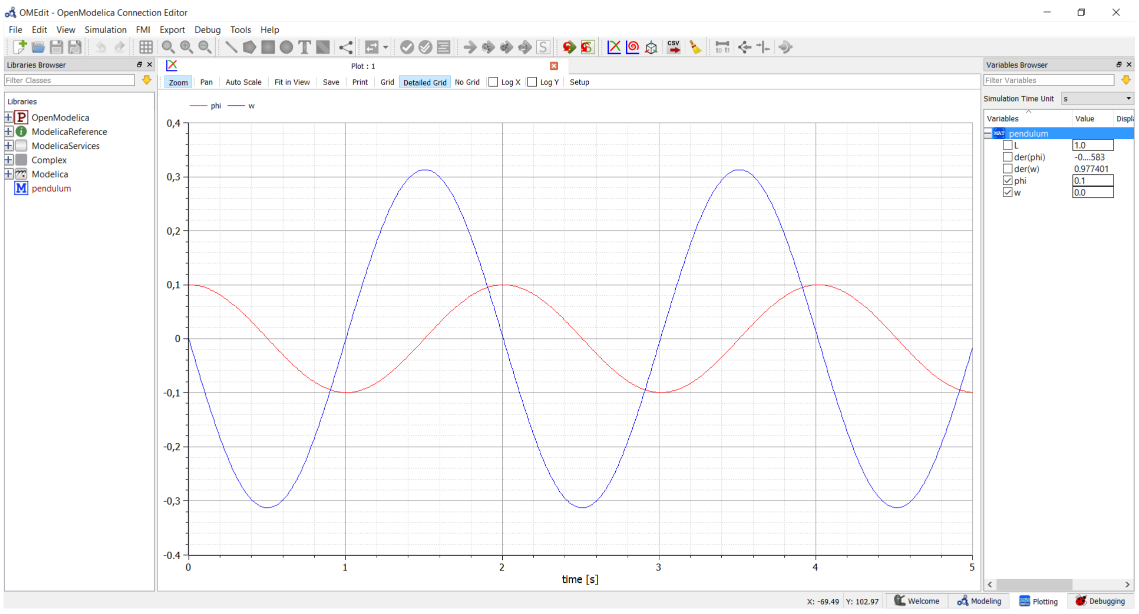
☐ Save experiment annotation inside model

☐ Save __OpenModelica_simulationFlags annotation inside model

☑ Simulate

OK | Cancel

Figure 1.12: Defining an experiment in OMEdit.



Variables Browser

Filter Variables

Simulation Time Unit: s

Variables	Value
pendulum	
☐ L	1.0
☐ der(phi)	-0....583
☐ der(w)	0.977401
☑ phi	0.1
☑ w	0.0

Figure 1.13: Running the simulation and plotting the results with OMEdit.

3. **Define the experiment**. By clicking on the *Simulation Setup* button (see the Figure 1.11), the window *Simulation Setup* is opened (see the Figure 1.12). In the *General* tab of this window, the user can set the initial and final time of the simulation, the length or number of output intervals, and the integration method and its tolerance. In this example, the final time is set to 5. The format of the file where the simulation results are stored can be set in the *Output* tab. Additionally, we can choose to store or not the *protected* type variables, the variable values at the event instants and the variable values at equidistant time grid.

4. **Run the simulation and plot the results**. The simulation starts by clicking on the button *Simulate* (see Figure 1.11). If an error occurs during the simulation run, OMEdit shows an error message in the message window. OMEdit shows the output variables in the window *Variables Browser* (see Figure 1.13). By clicking on one or several of these output variables, their graphical representation is displayed. Additionally, simulation results are stored in a file. OMEdit stores the output file in its working directory. The working directory can be changed in the *Options* window that is displayed by clicking on *Tools > Options*.

1.6 Further reading

Some of the ideas that motivated the Modelica standardization effort are explained by Åström et al. (1998), placing them in the context of the evolution of continuous-time modeling and simulation, since the mid 1920s until the late 1990s. We strongly recommend reading this article.

An outstanding book on the principles and techniques of continuous-time modeling is (Cellier 1991).

A selection of the connector variables was shown in Table 1.2, so that component connection satisfies power conservation. This concept is fully developed in the bond graph modeling formalism, according to which the system is modeled describing the flow, storage, dissipation and transformation of the energy that takes place in it. As these concepts are common to all physical domains, the bond graph formalism is well suited for mixed-domain modeling. Bond graph models reflects simultaneously the physical structure of the modeled system, and the computational causality of the model. Some excellent books on bond graph modeling are (Karnopp et al. 1990), (Thoma 1990) and (Hogan & Breedveld 1995).

The use of Dymola and OpenModelica was introduced in Section 1.5. Further information is provided in the Dymola documentation (Dynasim AB 2004, Dassault Systèmes AB 2016) and the OpenModelica website (OpenModelica 2017).

It is recommended to navigate around the Modelica Association website (ModelicaWebSite 2017), where there is plenty of information available: the successive Modelica language specifications, tutorials on the use of the language, conference papers, links to software tools, books, model libraries, etc.

Continuous-time atomic models

2.1 Introduction
2.2 Rectifier circuit
2.3 Translation in one dimension
2.4 Translation in two dimensions
2.5 Radial heat transfer in a pipe
2.6 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Declare scalar and array variables in Modelica.

– Relate basic types and attributes.

– Use the `Modelica.SIunits` and `Modelica.Constants` packages of the MSL.

– Declare and call functions in Modelica.

– Describe the continuous-time behavior using equations, algorithms and functions, involving scalar and array variables.

– Develop continuous-time atomic models in Modelica.

2.1 Introduction

This lesson provides an introduction to the description of continuous-time atomic models in Modelica. Atomic models are those models that are not composed of smaller components. The system behavior is described using equations and algorithms.

Only the description of continuous-time behavior is addressed in this lesson. Modeling of events will be discussed in the third part of the book, which is dedicated to hybrid system modeling and simulation.

Modeling examples in the electrical, mechanical and thermal domains are employed to explain and illustrate the concepts.

The rectifier circuit models described in Section 2.2 and the mechanical systems described in Section 2.3 are used to explain the declaration of scalar variables and their types; the meaning of the variable's attributes; the use of the standard types declared in the Modelica.SIunits package; and the basic structure of an atomic model, consisting in variable declarations and an equation section.

The declaration and use of vectors and matrices of variables is introduced in Section 2.4. To this end, a simplistic model of the Earth's motion around the Sun is developed. Two ways of describing the model behavior in terms of vector and matrix variables are discussed: employing vector and matrix equations, and for loops. The use of algorithms is introduced.

The example described in Section 2.5 is used to explain the definition and use of functions, which may facilitate the reuse of Modelica algorithms, and encapsulate calls to external functions. The example consists in modeling the stationary heat transfer in the radial direction of an insulated steel pipe.

2.2 Rectifier circuit

Consider the circuit shown in Figure 2.1, which is composed of a sinusoidal voltage generator, two resistors, a capacitor and a diode. These electronic components can be modeled with different level of detail, employing different types of mathematical model. We will model them using the constitutive relationships shown in Figure 2.2. This is, assuming that the generator, resistors and capacitor are ideal components, and the diode obeys the Shockley equation. Their parameter values are shown in Table 2.1. The circuit model can be built following the next steps.

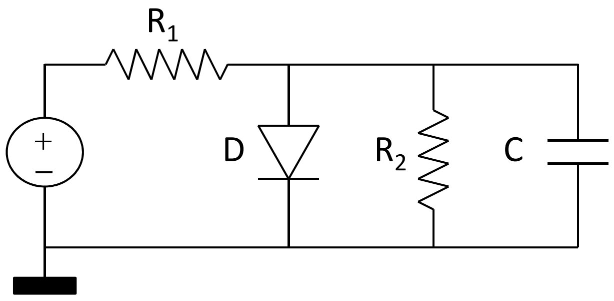


Figure 2.1: Electric circuit.

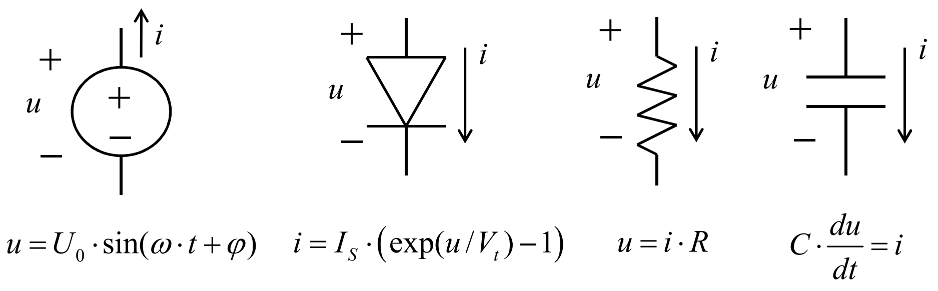

$u = U_0 \cdot \sin(\omega \cdot t + \varphi)$
$i = I_S \cdot \left(\exp(u/V_t) - 1\right)$
$u = i \cdot R$
$C \cdot \frac{du}{dt} = i$

Figure 2.2: Constitutive relationships of the components.

Table 2.1: Parameters of the circuit shown in Figure 2.1.

Component	**Parameter**	**Value**
Voltage generator	Amplitude	$U_0 = 5$ V
	Angular frequency	$w = 200\pi$ rad/s (= 100 Hz)
	Phase angle	$\varphi = 0$ rad
Resistors	Resistance	$R_1 = 100$ ohm
		$R_2 = 100$ ohm
Capacitor	Capacitance	$C = 10^{-6}$ F
Diode	Saturation current	$I_s = 10^{-9}$ A
	Thermal voltage	$V_t = 0.025$ V

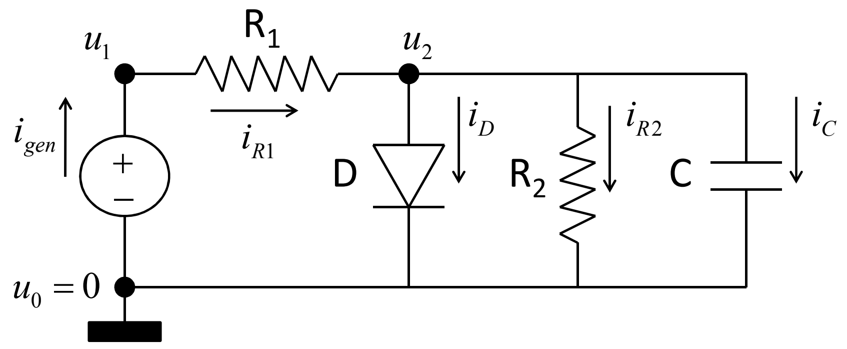


Figure 2.3: Assign names to the voltage nodes, and names and directions to the currents.

1. **Select a circuit node as reference for voltage**. This node is named **ground node** and by convention its voltage is zero. Voltage at the other circuit nodes is calculated relative to ground. The ground node has already been selected in the circuit shown in Figure 2.1.

2. **Assign names to the voltage in the remaining nodes**. Nodes are represented as filled circles in Figure 2.3. The names assigned are u_1 and u_2. The u_0 node is the ground node.

3. **Assign directions and names to the currents**. For each two-pin component, a name is given to the current that flows through the component with an arbitrarily chosen reference direction. In components with three or more pins, a name is assigned to the current in each pin and the current directions are chosen to be pointing towards the component. In this example, the names and directions given to the currents are shown in Figure 2.3. The names are: i_{gen}, i_{R1}, i_D, i_{R2}, i_C.

4. **Impose the current conservation in every node, except the ground node**. As charge is conserved in each node, the sum of currents at each node is zero. The equations at the nodes labeled as u_1 and u_2 are:

$$i_{gen} = i_{R1} \tag{2.1}$$

$$i_{R1} = i_D + i_{R2} + i_C \tag{2.2}$$

Observe that the equation that states the current conservation at the ground node does not provide any additional information and for this reason is not included in the model. This equation is always a linear combination of the equations that state the current conservation at the other nodes. For example, the equation at the ground node of this circuit is Eq. (2.3), which can be obtained adding Eqs. (2.1) and (2.2).

$$i_{gen} = i_D + i_{R2} + i_C \tag{2.3}$$

5. **Write the constitutive relationships of the components**. The constitutive relationships of the components that intervene in this circuit are shown in Figure 2.2, where u represents the voltage drop across the component. This is, the voltage at the pin labeled "+" minus the voltage at the pin labeled "-". The equations describing the five components of the circuit are:

$$\begin{aligned}
u_1 - u_0 &= U_0 \cdot \sin(w \cdot t + \varphi) && (2.4)\\
u_1 - u_2 &= i_{R1} \cdot R_1 && (2.5)\\
i_D &= I_s \cdot \left(exp\left(\frac{u_2 - u_0}{V_t} \right) - 1 \right) && (2.6)\\
u_2 - u_0 &= i_{R2} \cdot R_2 && (2.7)\\
C \cdot \frac{d}{dt}(u_2 - u_0) &= i_C && (2.8)
\end{aligned}$$

Replacing the ground node voltage (u_0) by zero, it is obtained:

$$\begin{aligned}
u_1 &= U_0 \cdot \sin(w \cdot t + \varphi) && (2.9)\\
u_1 - u_2 &= i_{R1} \cdot R_1 && (2.10)\\
i_D &= I_s \cdot \left(exp\left(\frac{u_2}{V_t} \right) - 1 \right) && (2.11)\\
u_2 &= i_{R2} \cdot R_2 && (2.12)\\
C \cdot \frac{du_2}{dt} &= i_C && (2.13)
\end{aligned}$$

The model is composed by the equations describing the current conservation at the nodes, Eqs. (2.1) and (2.2), and the constitutive relationships of the components, Eqs. (2.9) – (2.13). These seven equations describe the time evolution of the voltages (u_1, u_2) and currents (i_{gen}, i_{R1}, i_D, i_{R2}, i_C). The parameter values are given in Table 2.1.

The circuit model can be described in Modelica as shown in Modelica Code 2.1. Observe the following::

```
model circuit1
  Real i_gen(unit="A") "Current of the generator";
  Real i_R1(unit="A") "Current of R1";
  Real i_R2(unit="A") "Current of R2";
  Real i_C(unit="A") "Current of the capacitor";
  Real i_D(unit="A") "Current of the diode";
  Real u_1(unit="V") "Voltage of generator";
  Real u_2(start=0, fixed=true, unit="V") "Output voltage";
  // Voltage generator
  constant Real PI = 3.1415926536;
  parameter Real U0( unit="V") = 5;
  parameter Real frec( unit="Hz") = 100;
  parameter Real w( unit="rad/s") = 2*PI*frec;
  parameter Real phi( unit="rad") = 0;
  // Resistors
  parameter Real R1( unit="ohm") = 100;
  parameter Real R2( unit="ohm") = 100;
  // Capacitor
  parameter Real C( unit="F") = 1e-6;
  // Diode
  parameter Real Is( unit="A") = 1e-9;
  parameter Real Vt( unit="V") = 0.025;

equation
  // Node equations
  i_gen = i_R1;
  i_R1 = i_D + i_R2 + i_C;
  // Constitutive relationships
  u_1 = U0 * sin( w * time + phi);
  u_1 - u_2 = i_R1 * R1;
  i_D = Is * ( exp(u_2 / Vt) - 1);
  u_2 = i_R2 * R2;
  C * der(u_2) = i_C;
end circuit1;
```

Modelica Code 2.1: Circuit of Figure 2.1, with the parameters of Table 2.1.

– Declaring a variable implies specifying its data type (Real, Integer, Boolean, etc.), its variability (see Table 2.2), name and dimensionality (scalar, n-dimensional matrix). Optionally, values can be assigned to the variable attributes at the variable declaration. The attributes associated to a variable depends on the variable data type (see Tables 2.3 and 2.4).

– The constitutive relationship of the generator describes a sinusoidal voltage waveform, in which time appears explicitly. Time is represented by a built-in Modelica variable named **time**.

– The **start** and **fixed** attributes have been employed to specify the initial value of the u_2 variable. This initial value specified in the variable declaration can be later modified in the experiment definition.

– The units have been indicated in the variable declarations, using the **unit** attribute.

Table 2.2: Variability of variables.

Keyword	**Meaning**
constant	The value of a constant variable cannot be modified at the experiment definition and does not change during the simulation.
parameter	The value of a parameter can be modified when the component is instantiated or inherited, and at the experiment definition, but parameter values don't change during the simulation.
discrete	It is optional and indicates that the variable is a discrete-time variable. As modeling environments are able to automatically deduce whether a variable is continuous-time or discrete-time, this keyword is seldom used. Variables of *Integer*, *Boolean* or *String* type are discrete-time, and the use of *Real* variables indicates whether are continuous-time or discrete-time.
(no keyword)	Continuous-time or discrete-time variables.

Table 2.3: Attributes of variables.

Attribute	**Meaning**
quantity	Physical quantity (e.g., `quantity = "Time"`, `quantity="Mass"`).
unit	Units of the physical quantity (e.g., `unit = "s"`, `unit = "kg"`).
min, *max*	Minimum and maximum values allowed for the variable. The variable value going out of this range indicates that the model is not reproducing the behavior of the real system. Some modeling environments generate warning messages during the simulation when variables go out of their range. The *assert* sentence can be used for the same purpose.
start, *fixed*	The *start* and *fixed* attributes are related. If *fixed* is *true*, the *start* value is the variable initial value. If *fixed* is *false* and the variable is calculated at the initial time solving a system of simultaneous equations, then the *start* value is used by the numerical iterative method as initial guess for calculating the variable at the initial time.
displayUnit	Units employed by the graphical user interface of the modeling environment for displaying the results.

Table 2.4: Attributes of the built-in types of variable.

	Real	*Integer*	*Boolean*	*String*
quantity	×	×	×	×
unit	×			
min, *max*	×	×		
start, *fixed*	×	×	×	×
displayUnit	×			

```
package SIunits "Type definitions based on SI units, ISO 31-1992"
   ...
  // Space and Time (chapter 1 of ISO 31-1992)
  type Angle = Real(final quantity="Angle",final unit="rad",displayUnit="deg");
  type Frequency = Real(final quantity="Frequency", final unit="Hz");
  type AngularFrequency = Real(final quantity="AngularFrequency",final unit="s-1");
  ...
  // Electricity and Magnetism (chapter 5 of ISO 31-1992)
  type ElectricCurrent = Real(final quantity="ElectricCurrent", final unit="A");
  type Current = ElectricCurrent;
  type ElectricPotential = Real(final quantity="ElectricPotential", final unit="V");
  type Voltage = ElectricPotential;
  type Capacitance = Real(final quantity="Capacitance",final unit="F",min=0);
  type Resistance = Real(final quantity="Resistance",final unit="Ohm");
    ...
end SIunits;
```

Modelica Code 2.2: Fragment of the Modelica.SIunits package.

Modelica facilitates declaring new variable types, based on the built-in types, by setting attribute values. For instance, two new types named `Voltage` and `Current` can be declared as follows:

```
type Voltage  = Real( unit = "V" );
type Current  = Real( unit = "A" );
```

The Modelica Standard Library (MSL) includes the **Modelica.SIunits** package that contains type declarations following the International System of Units. This package is intended to standardize the declaration of physical quantity types and therefore the use of these types is recommended. A fragment of this package is shown in Modelica Code 2.2. The **final** keyword written before an attribute indicates that further modifications in the value of this attribute are not allowed.

Dot notation is used for accessing to the classes defined within a package. The model shown in Modelica Code 2.1 can be rewritten using the types of the Modelica.SIunits package as shown in Modelica Code 2.3. Observe that, instead of declaring the `PI` constant, it has been used a constant named `pi` that is declared in the **Modelica.Constants** package of the MSL. The evolution of the node voltages shown in Figure 2.4 is obtained simulating Modelica Code 2.3 from time 0 s until time 0.05 s.

Connecting three additional diodes to the circuit in Figure 2.1 and removing one resistor, it is obtained the rectifier circuit shown in Figure 2.5. The ground node has been selected, names have been assigned to the node voltages, and names and directions have been assigned to the currents that flow through the components.

```
model circuit1
  Modelica.SIunits.Current i_gen "Current of the generator";
  Modelica.SIunits.Current i_R1 "Current of R1";
  Modelica.SIunits.Current i_R2 "Current of R2";
  Modelica.SIunits.Current i_C "Current of the capacitor";
  Modelica.SIunits.Current i_D "Current of the diode";
  Modelica.SIunits.Voltage u_1 "Voltage of generator";
  Modelica.SIunits.Voltage u_2(start=0, fixed=true) "Output voltage";
  // Voltage generator
  parameter Modelica.SIunits.Voltage U0 = 5;
  parameter Modelica.SIunits.Frequency frec = 100;
  parameter Modelica.SIunits.AngularFrequency w = 2*Modelica.Constants.pi*frec;
  parameter Modelica.SIunits.Angle phi = 0;
  // Resistors
  parameter Modelica.SIunits.Resistance R1 = 100;
  parameter Modelica.SIunits.Resistance R2 = 100;
  // Capacitor
  parameter Modelica.SIunits.Capacitance C = 1e-6;
  // Diode
  parameter Modelica.SIunits.Current Is = 1e-9;
  parameter Modelica.SIunits.Voltage Vt = 0.025;

equation
  // Node equations
  i_gen = i_R1;
  i_R1 = i_D + i_R2 + i_C;
  // Constitutive relationships
  u_1 = U0 * sin( w * time + phi);
  u_1 - u_2 = i_R1 * R1;
  i_D = Is * ( exp(u_2 / Vt) - 1);
  u_2 = i_R2 * R2;
  C * der(u_2) = i_C;
end circuit1;
```

Modelica Code 2.3: Circuit of Figure 2.1, described using the Modelica.SIunits types.

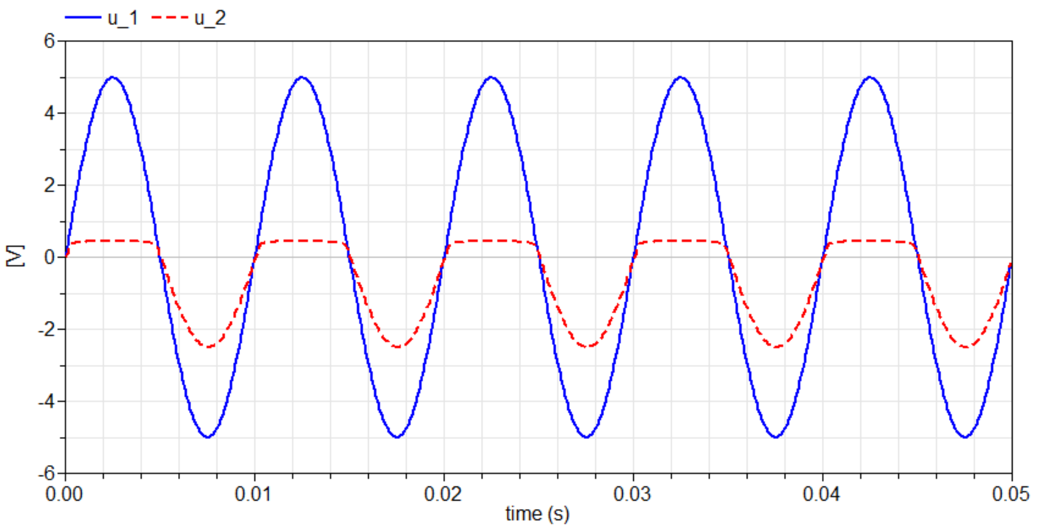


Figure 2.4: Voltages of the circuit shown in Figure 2.1 obtained simulating the Modelica Code 2.3.

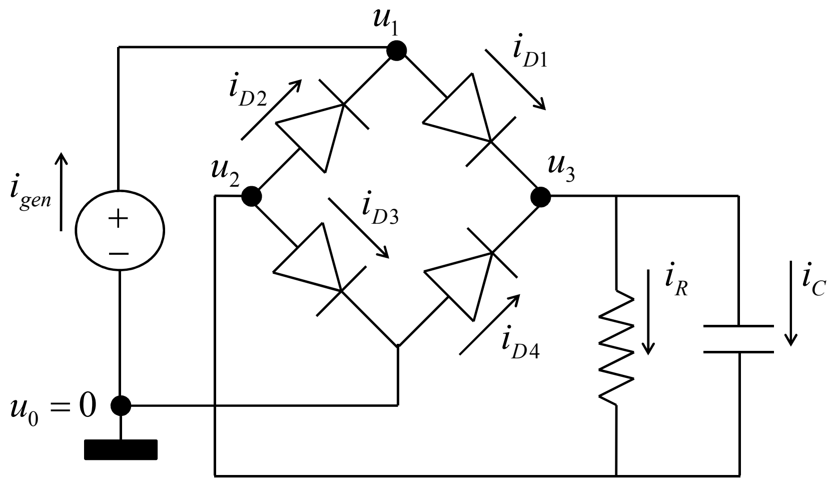


Figure 2.5: Rectifier circuit with diode bridge.

The model is composed of the current conservation equations and the constitutive relationships of the components. Assuming that the ground node voltage is zero, the model is described by Eqs. (2.14) – (2.24).

$$i_{gen} + i_{D2} = i_{D1} \tag{2.14}$$
$$i_R + i_C = i_{D2} + i_{D3} \tag{2.15}$$
$$i_{D1} + i_{D4} = i_R + i_C \tag{2.16}$$
$$u_1 = U_0 \cdot \sin(w \cdot t + \varphi) \tag{2.17}$$
$$i_{D1} = I_s \cdot \left(exp\left(\frac{u_1 - u_3}{V_t} \right) - 1 \right) \tag{2.18}$$
$$i_{D2} = I_s \cdot \left(exp\left(\frac{u_2 - u_1}{V_t} \right) - 1 \right) \tag{2.19}$$
$$i_{D3} = I_s \cdot \left(exp\left(\frac{u_2}{V_t} \right) - 1 \right) \tag{2.20}$$
$$i_{D4} = I_s \cdot \left(exp\left(\frac{-u_3}{V_t} \right) - 1 \right) \tag{2.21}$$
$$u_3 - u_2 = i_R \cdot R \tag{2.22}$$
$$u_C = u_3 - u_2 \tag{2.23}$$
$$C \cdot \frac{du_C}{dt} = i_C \tag{2.24}$$

The model description is shown in Modelica Code 2.4. The generator (u_1) and output (u_C) voltages obtained simulating the model during 0.05 s are displayed in Figure 2.6.

```
 model circRectifier
  Modelica.SIunits.Current i_gen "Current of the generator";
  Modelica.SIunits.Current i_R "Current through the resistor";
  Modelica.SIunits.Current i_C "Capacitor current";
  Modelica.SIunits.Current i_D1 "Current through diode 1";
  Modelica.SIunits.Current i_D2 "Current through diode 2";
  Modelica.SIunits.Current i_D3 "Current through diode 3";
  Modelica.SIunits.Current i_D4 "Current through diode 4";
  Modelica.SIunits.Voltage u_1 "Voltage of generator";
  Modelica.SIunits.Voltage u_2 "Voltage at node u2";
  Modelica.SIunits.Voltage u_3 "Voltage at node u3";
  Modelica.SIunits.Voltage u_C(start=0, fixed=true) "Capacitor voltage";
  // Generator parameters
  parameter Modelica.SIunits.Voltage U0=5;
  parameter Modelica.SIunits.Frequency frec=100;
  parameter Modelica.SIunits.AngularFrequency w=2*Modelica.Constants.pi*frec;
  parameter Modelica.SIunits.Angle phi=0;
  // Resistor
  parameter Modelica.SIunits.Resistance R=100;
  // Capacitor
  parameter Modelica.SIunits.Capacitance C=1e-6;
  // Diodes
  parameter Modelica.SIunits.Current Is=1e-9;
  parameter Modelica.SIunits.Voltage Vt=0.025;

equation
  // Current conservation equations
  i_gen + i_D2 = i_D1;
  i_R + i_C = i_D2 + i_D3;
  i_D1 + i_D4 = i_R + i_C;
  // Constitutive relationships
  u_1 = U0*sin(w*time + phi);
  i_D1 = Is*(exp((u_1 - u_3)/Vt) - 1);
  i_D2 = Is*(exp((u_2 - u_1)/Vt) - 1);
  i_D3 = Is*(exp(u_2/Vt) - 1);
  i_D4 = Is*(exp(-u_3/Vt) - 1);
  u_3 - u_2 = i_R*R;
  u_C = u_3 - u_2;
  C*der(u_C) = i_C;
end circRectifier;
```

Modelica Code 2.4: Model of the rectifier circuit shown in Figure 2.5.

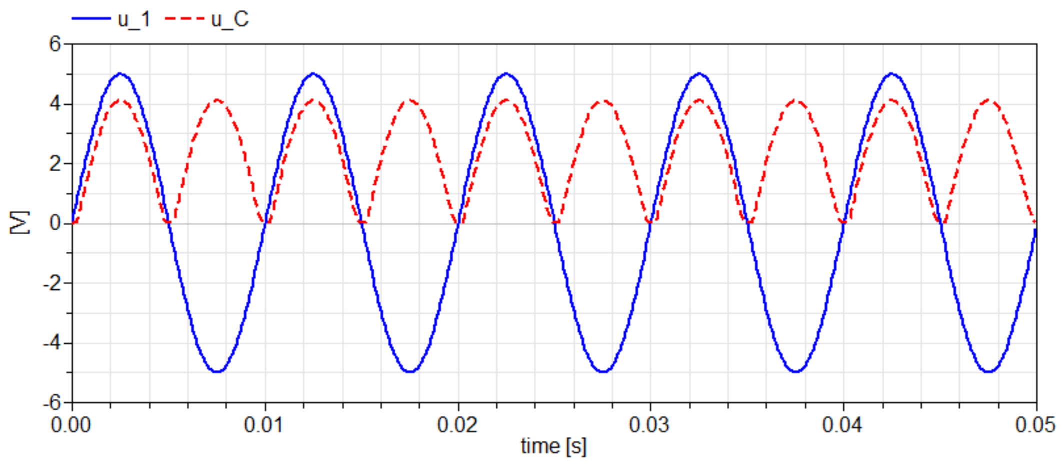


Figure 2.6: Result obtained of simulating Modelica Code 2.4.

2.3 Translation in one dimension

Let's consider the system represented in Figure 2.7. It is composed of a first body that slides on a horizontal surface, and a second body that slides on the first one. The two movements are one dimensional and have the same direction. There exists a friction force between the first body and the surface, and between the two bodies. It is assumed that this friction force is proportional to the relative velocity of the two contacting materials. The body masses, m_1 and m_2, are constant. The objective is to develop a model that describes the evolution of the body positions and velocities.

The first step in developing the model is to choose a reference and sign convention for velocity. We choose the velocity of the horizontal surface as reference, and assign it the zero value. The body velocities with respect to the horizontal surface are v_1 and v_2, respectively. We choose the following sign convention for velocity: positive if the body moves towards right, and negative if moves towards left.

The evolution of the body velocities can be calculated applying the second Law of Newton. According to this law, the change in the linear momentum of a body produced by a net force is directly proportional to the magnitude of the net force F, in the same direction as the net force.

$$\frac{d}{dt}(m \cdot v) = F \tag{2.25}$$

When the mass of the body is constant, Eq. (2.25) can be written as follows:

$$m \cdot \frac{dv}{dt} = F \tag{2.26}$$

Let's write Eq. (2.26) particularized for each of the two bodies that compose our system. Forces of friction are exerted on the first body by the horizontal surface and the second body. The force exerted on the second body is due to the friction with the first body.

$$m_1 \cdot \frac{dv_1}{dt} = \underbrace{-b_{s,1} \cdot v_1}_{\substack{\text{Force exerted by the} \\ \text{horizontal surface on body 1}}} \quad \underbrace{-b_{1,2} \cdot (v_1 - v_2)}_{\substack{\text{Force exerted by} \\ \text{body 2 on body 1}}} \tag{2.27}$$

$$m_2 \cdot \frac{dv_2}{dt} = \underbrace{-b_{1,2} \cdot (v_2 - v_1)}_{\substack{\text{Force exerted by} \\ \text{body 1 on body 2}}} \tag{2.28}$$

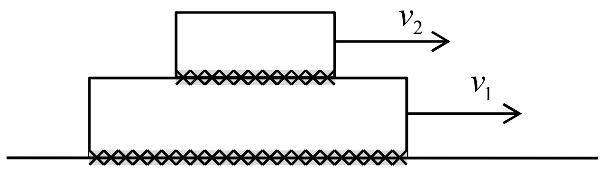


Figure 2.7: Movement of two bodies with friction.

```
model TwoBodiesWithFriction
  import SI = Modelica.SIunits;
  parameter SI.Mass m1=100 "Mass of body 1";
  parameter SI.Mass m2=10 "Mass of body 2";
  parameter SI.TranslationalDampingConstant b_s1=1.8
    "Friction coefficient of surface-body 1";
  parameter SI.TranslationalDampingConstant b_12=1.6
    "Friction coefficient of body 1-body 2";
  SI.Position x1(start=0, fixed=true) "Position of body 1";
  SI.Position x2(start=0, fixed=true) "Position of body 2";
  SI.Velocity v1(start=0.1, fixed=true) "Velocity of body 1";
  SI.Velocity v2(start=0, fixed=true) "Velocity of body 2";
equation
  // Body 1
  der(x1) = v1;
  m1*der(v1) = -b_s1*v1 - b_12*(v1 - v2);
  // Body 2
  der(x2) = v2;
  m2*der(v2) = -b_12*(v2 - v1);
end TwoBodiesWithFriction;
```

Modelica Code 2.5: Model of the two-body system shown in Figure 2.7.

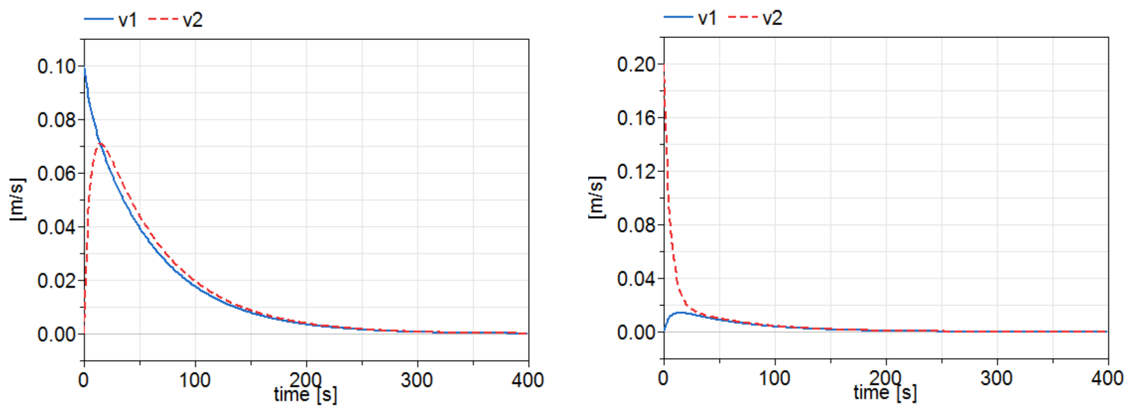


Figure 2.8: Simulation of Modelica Code 2.5 with two different initial conditions: $v_1(0) = 0.1$ m/s, $v_2(0) = 0$ (left); $v_1(0) = 0$, $v_2(0) = 0.2$ m/s (right).

The $b_{s,1}$ and $b_{1,2}$ parameters are the proportionality coefficients that relate the force of friction and the relative velocities.

Observe that, if the body velocities are known at the initial time, Eqs. (2.27) and (2.28) allow to calculate the velocities in the successive time instants.

The next step in developing the model is to include the equations that relate the body positions with their velocities. To this end, we have to choose a reference and a sign convention for the spatial coordinate. The sign criterion for position is usually chosen in consonance with the sign criterion for velocity, so that the value of the spatial coordinate increases if the velocity is positive, and decreases if negative. As the velocity is positive if the body moves towards right, the spatial coordinate of our model increases towards right. In this way, the following relationships hold between velocities (v_1, v_2) and positions (x_1, x_2):

$$\frac{dx_1}{dt} = v_1 \tag{2.29}$$

$$\frac{dx_2}{dt} = v_2 \tag{2.30}$$

Known the body positions at the initial time, the evolution of the positions can be calculated from Eqs. (2.29) and (2.30).

Setting an initial value for the body position implies choosing the origin of the coordinate system. This should be done in a way that facilitates as much as possible the interpretation of the simulation results. For instance, the origin of the coordinate system for both bodies could be a certain location in the horizontal surface. Other option would be to assign the zero value to the initial position of the body, so that the position of each body measures its displacement with respect to its initial position.

The system model is described by Eqs. (2.27)–(2.30). Setting the constant values of the body masses (m_1, m_2) and the friction coefficients ($b_{s,1}$, $b_{1,2}$), and the initial values of the body positions and velocities, the model allows to calculate the evolution of the body positions and velocities.

The Modelica description of the model is shown in Modelica Code 2.5. The evolutions of the body velocities for two different initial conditions are shown in Figure 2.8. The observed behavior is explained considering that the force of friction exerted by the horizontal surface makes the velocity of the first body to tend to zero, and the force of friction between the two bodies makes their relative velocity to tend to zero.

In the first simulation run, the first body is initially moving and the second body is at rest. The force of friction between the bodies makes the second body to start moving, increasing its velocity. When the relative velocity between the bodies is zero, the friction force between them vanishes. However, the horizontal surface continues braking the first body and consequently reducing its velocity. When the velocity of the first body becomes smaller than the velocity of the second body, the force of friction between the bodies acts again, but in this case reducing the velocity of the second body.

In the second simulation run, the first body is initially at rest and is accelerated by the force of friction exerted by the second body.

Let's make the system a little more complex by adding some components. Consider the system depicted in Figure 2.9. It consists of two bodies with constant masses, m_1 and m_2, moving on a horizontal surface, and a third body, with constant mass m_3, that moves on them. There is friction between all contacting materials. A damper connects the first and second bodies, which are connected by springs to the lateral walls. A given sinusoidal force, F, is applied to the third body. The horizontal surface and the lateral walls remain at rest. The model should describe the evolution of the velocity and position of the three bodies.

The first step in developing the model consists in choosing the reference and sign convention for velocity. We choose as a reference the velocity of the horizontal surface and the lateral walls, and assign to this velocity the value zero. In addition, we consider that a body moving towards right has positive velocity. We adopt the same sign convention for the force: it is positive if is pointing towards right.

The second Newton's Law allows to relate the changes in the linear momentum of each body with the net force applied on it.

$$m_1 \cdot \frac{dv_1}{dt} = \underbrace{-b_1 \cdot v_1}_{\substack{\text{Force of friction exerted}\\ \text{by horizontal surface}}} \underbrace{-b_4 \cdot (v_1 - v_3)}_{\substack{\text{Force of friction}\\ \text{exerted by body 3}}} \underbrace{-b_2 \cdot (v_1 - v_2)}_{\substack{\text{Force of}\\ \text{dumper}}} \underbrace{-k_1 \cdot e_1}_{\substack{\text{Force of}\\ \text{spring 1}}} \tag{2.31}$$

$$m_2 \cdot \frac{dv_2}{dt} = \underbrace{-b_3 \cdot v_2}_{\substack{\text{Force of friction exerted}\\ \text{by horizontal surface}}} \underbrace{-b_5 \cdot (v_2 - v_3)}_{\substack{\text{Force of friction}\\ \text{exerted by body 3}}} \underbrace{-b_2 \cdot (v_2 - v_1)}_{\substack{\text{Force of}\\ \text{dumper}}} \underbrace{+k_2 \cdot e_2}_{\substack{\text{Force of}\\ \text{spring 2}}} \tag{2.32}$$

$$m_3 \cdot \frac{dv_3}{dt} = \underbrace{-b_4 \cdot (v_3 - v_1)}_{\substack{\text{Force of friction}\\ \text{exerted by body 1}}} \underbrace{-b_5 \cdot (v_3 - v_2)}_{\substack{\text{Force of friction}\\ \text{exerted by body 2}}} \underbrace{+F}_{\substack{\text{External}\\ \text{force}}} \tag{2.33}$$

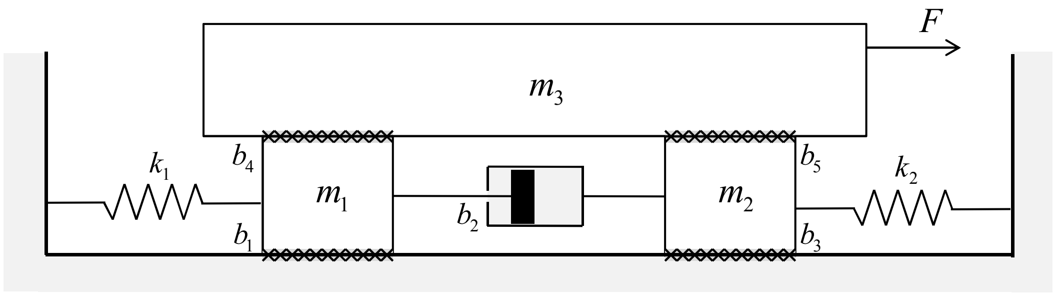


Figure 2.9: Movement of three bodies with friction, connected by dumper and springs.

```
model ThreeBodiesWithFriction
  import SI = Modelica.SIunits;
  parameter SI.Mass m1=100 "Mass of body 1";
  parameter SI.Mass m2=10  "Mass of body 2";
  parameter SI.Mass m3=15  "Mass of body 3";
  parameter SI.TranslationalDampingConstant b1=1.8 "Body 1 - Surface";
  parameter SI.TranslationalDampingConstant b2=3.2 "Body 1 - Body 2";
  parameter SI.TranslationalDampingConstant b3=1.8 "Body 2 - Surface";
  parameter SI.TranslationalDampingConstant b4=1.6 "Body 1 - Body 3";
  parameter SI.TranslationalDampingConstant b5=1.6 "Body 2 - Body 3";
  parameter SI.TranslationalSpringConstant k1=1.2 "Spring 1";
  parameter SI.TranslationalSpringConstant k2=1.2 "Spring 2";
  parameter SI.Force F0=10 "Amplitude of the external force";
  parameter SI.AngularFrequency w=0.1 "Frequency of the external force";
  SI.Position x1(start=0, fixed=true) "Position of body 1";
  SI.Position x2(start=0, fixed=true) "Position of body 2";
  SI.Position x3(start=0, fixed=true) "Position of body 3";
  SI.Velocity v1(start=0, fixed=true) "Velocity of body 1";
  SI.Velocity v2(start=0, fixed=true) "Velocity of body 2";
  SI.Velocity v3(start=0, fixed=true) "Velocity of body 3";
  SI.Length e1(start=1, fixed=true) "Elongation of spring 1";
  SI.Length e2(start=0, fixed=true) "Elongation of spring 2";
  SI.Force F "External force applied on body 3";
equation
  // Body 1
  der(x1) = v1;
  m1*der(v1) = -b1*v1 - b4*(v1 - v3) - b2*(v1 - v2) - k1*e1;
  // Body 2
  der(x2) = v2;
  m2*der(v2) = -b3*v2 - b5*(v2 - v3) - b2*(v2 - v1) + k2*e2;
  // Body 3
  der(x3) = v3;
  m3*der(v3) = -b4*(v3 - v1) - b5*(v3 - v2) + F;
  // Spring 1
  der(e1) = v1;
  // Spring 2
  der(e2) = -v2;
  // External force
  F = F0*sin(w*time);
end ThreeBodiesWithFriction;
```

Modelica Code 2.6: Model of the system depicted in Figure 2.9.

where e represents the difference between the actual length of the spring and its natural length. The left end of the first spring is at rest, while the right end moves with velocity v_1. The right end of the second spring is at rest and the left end moves with velocity v_2. Therefore:

$$\frac{de_1}{dt} = v_1 \tag{2.34}$$
$$\frac{de_2}{dt} = -v_2 \tag{2.35}$$

The relationship between the position and velocity of each body can be expressed as follows:

$$\frac{dx_1}{dt} = v_1 \tag{2.36}$$
$$\frac{dx_2}{dt} = v_2 \tag{2.37}$$
$$\frac{dx_3}{dt} = v_3 \tag{2.38}$$

The system model consists of Eqs. (2.31) – (2.38), and the constant known values of the body masses (m_1, m_2, m_3), the spring coefficients (k_1, k_2), the dumper coefficient (b_2) and the friction coefficients (b_1, b_3, b_4, b_5).

For setting the initial state of the system, we will give an initial value to the position and velocity of each body, and to the difference between the actual length and the natural length of each spring.

As origin of the coordinate system for the displacement of each body, we chose the body position at the initial time. Therefore, the initial position of the three bodies is zero. In this way, the x variable of each body indicates its displacement with respect to its initial position. The Modelica description of the model is listed in Modelica Code 2.6.

2.4 Translation in two dimensions

A simplistic model of the Earth's motion around the Sun will allow us to introduce the use of vectors and matrices in Modelica. Let's assume that the Earth orbits in a plane. In this plane, we define a two-dimensional rectangular coordinate system.

The origin of this coordinate system is at the center of the Sun. We represent the position and velocity of the Earth using two two-dimensional vectors named $\mathbf{x}$ and $\mathbf{v}$ respectively (see Figure 2.10).

The attractive force between two bodies with masses M and m can be calculated by applying Eq. (2.39), where G is the gravitational constant. This equation is known as the Newton's law of universal gravitation. The second Newton's Law relates the force of gravity and the acceleration of the Earth around the Sun, as shown in Eq. (2.40). The relationship between velocity and position is given by Eq. (2.41).

$$\mathbf{F} = -G \cdot \frac{M \cdot m}{\mathbf{x} \cdot \mathbf{x}} \cdot \frac{\mathbf{x}}{|\mathbf{x}|} \tag{2.39}$$

$$m \cdot \frac{d\mathbf{v}}{dt} = \mathbf{F} \tag{2.40}$$

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \tag{2.41}$$

Let's give the following values to the Earth's mass (m) and Sun's mass (M): $m = 5.976 \cdot 10^{24}$ kg and $M = 1.989 \cdot 10^{30}$ kg. The initial state of the system is specified by setting the initial position and velocity of the Earth. Let's suppose that the Earth is initially at the $\mathbf{x}(0) = \{152.1 \cdot 10^9, 0\}$ m position and is moving with a velocity of $\mathbf{v}(0) = \{0, 29.29 \cdot 10^3\}$ m/s.

The model is described in Modelica Code 2.7. Two-component vectors (one-dimensional arrays) are employed to represent the position, velocity and force. The model equations, Eqs. (2.39) – (2.41), are described in Modelica using vector equations. The Earth trajectory obtained simulating this model during one year (=3.1536e7 s) is shown in Figure 2.11.

An **array variable** is declared indicating between square brackets the number of its components in each dimension. This size declaration can be indistinctly written after the variable type or after the variable name. For instance,

```
Real[2] x, v, F;            |      Real x[2], v[2], F[2];
Real[6,2,10] tabla;         |      Real tabla[6,2,10];
```

where `r`, `v` and `F` are **vector variables** (one-dimensional arrays) with two components, and `tabla` is a three-dimensional **matrix variable** (array with dimension greater than one), with 6-by-2-by-10 components. The declaration of array variables can also be made indicating only the number of dimensions, for instance,

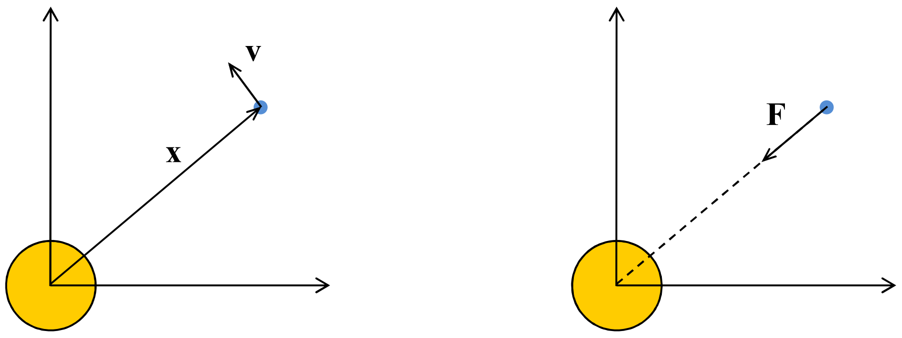


Figure 2.10: Position and velocity of the Earth (left); and gravitational force exerted by the Sun on the Earth (right).

```
model SunEarthSystem
  import SI = Modelica.SIunits;
  SI.Position x[2](start={152.1e9,0}, fixed=true) "Earth position";
  SI.Velocity v[2](start={0,29.29e3}, fixed=true) "Earth velocity";
  SI.Force F[2] "Gravitational force";
  parameter SI.Mass m=5.976e24 "Mass of the Earth";
  parameter SI.Mass M=1.989e30 "Mass of the Sun";
equation
  // Gravitational force exerted by the Sun on the Earth
  F = -(Modelica.Constants.G*m*M/(x*x))*(x/sqrt(x*x));
  // Position and velocity of the Earth
  m*der(v) = F;
  der(x) = v;
end SunEarthSystem;
```

Modelica Code 2.7: Model of the Sun-Earth system.

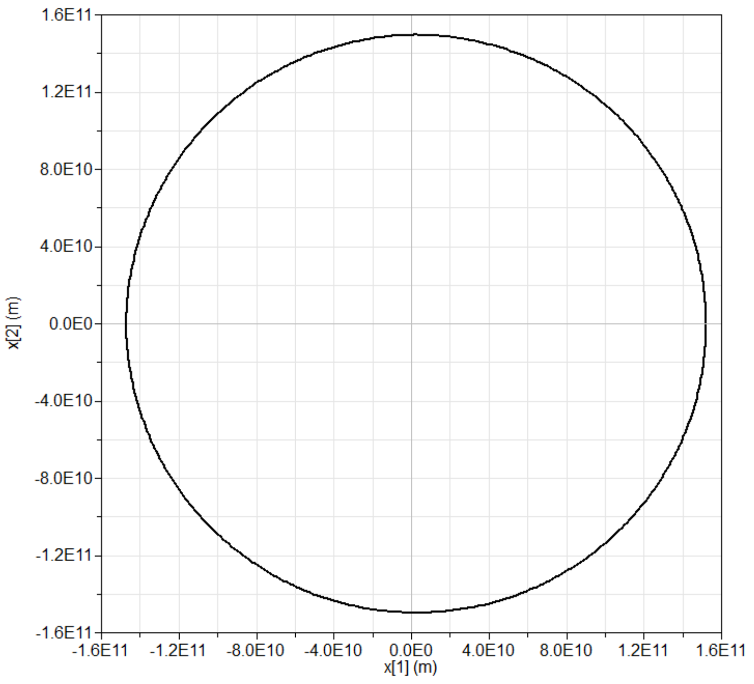


Figure 2.11: Earth's orbit obtained simulating the Modelica Code 2.7.

```
Real[:] x, v, F;            |   Real x[:], v[:], F[:];
Real[:,:,:] tabla;          |   Real tabla[:,:,:];
```

nevertheless, the model must contain the necessary information for the modeling environment to calculate the number of components. The modeling environment calculates the number of components by analyzing the equations in which the array variables intervene.

In any case, the number of dimensions and the number of components in each dimension cannot change during the simulation run.

An array component can be referenced specifying its index, whose numbering **starts at one**. For instance, `x[i]` is the i-th component of the vector. If `x` has two components, these are `x[1]` and `x[2]`.

Sub-arrays can be specified indicating a **range of indexes**. For instance, `A[i1:i2, j1:j2]` is an array composed of the `i1`-to-`i2` rows and the `j1`-to-`j2` columns of `A`.

Some built-in functions useful for working with arrays are listed in Table 2.5. Examples of array equations are provided in Table 2.6.

Observe in Modelica Code 2.7 that the initial values have been set by giving value to the *start* attributes. Vectors are written in this example as comma-separated lists inside curly braces. Some examples showing how to assign the value of vectors and matrices can be found in Table 2.7. The array value can also be specified using a **for expression with iterators**, whose syntax is basically: `{ expression for iterators }`. See the examples in Table 2.8.

As shown in Modelica Code 2.7, the model of the Sun-Earth system has been written using vector equations:

$\mathbf{F} = -G \cdot \frac{m \cdot M}{\mathbf{x} \cdot \mathbf{x}} \cdot \frac{\mathbf{x}}{|\mathbf{x}|}$ $\rightarrow$ `F = - Modelica.Constants.G * m * M / (x*x) * x / sqrt(x*x);`

$m \cdot \frac{d\mathbf{v}}{dt} = \mathbf{F}$ $\rightarrow$ `m * der(v) = F;`

$\frac{d\mathbf{x}}{dt} = \mathbf{v}$ $\rightarrow$ `der(x) = v;`

For-clauses and if-clauses can also be employed for writing array equations and assignments. The **for-clauses** are translated by the modeling environment into a set of equations or a sequence of assignments. The for-clause syntax is basically (see examples in Tables 2.9 and 2.10):

```
for for_indices loop
   ...
end for;
```

Table 2.5: Some built-in functions for arrays.

Function call	**Returned value**
`ndims(A)`	Number of dimensions of A.
`size(A, i)`	Number of components of A in the i-th dimension.
`size(A)`	Vector of length ndims(A) whose components are the size of A in each dimension.
`diagonal(v)`	Square diagonal matrix with the elements of vector v on the main diagonal.
`fill(s,n1,n2,...)`	$n_1 \times n_2 \times \ldots$ matrix filled with the s value.
`transpose(A)`	Array obtained by permuting the two first dimensions of A.
`zeros(n1,n2,...)`	$n_1 \times n_2 \times \ldots$ array filled with zeros.
`ones(n1,n2,...)`	$n_1 \times n_2 \times \ldots$ array filled with ones.
`identity(n)`	$n \times n$ identity matrix.
`linspace(x1,x2,n)`	n-component vector with evenly spaced values between x_1 and x_2.
`min(A), max(A)`	Lowest/Largest value in A.
`sum(A), product(A)`	Sum/Product of the components of A.
`cross(x,y)`	Cross product of the 3-component vectors x and y.

Table 2.6: Examples of array equations.

Fragment of Modelica code	**Comment to the code**

`   Real A[3,2], B[2,4];`
`   Real C[3,4];`
`equation`
`   C = A * B;`
	As A, B and C are matrices, the $*$ operator is interpreted as matrix multiplication.

`   Real x[3], y[3];`
`   Real z;`
`equation`
`   z = x * y;`
	As x and y are vectors, and z is scalar, the $*$ operator is interpreted as dot product.

`   Real x[3], y[3];`
`   Real z[3];`
`equation`
`   z = cross (x,y);`
	The *cross* function returns the cross product of the two 3-component vectors x and y (see Table 2.5).

Table 2.7: Examples showing how to declare and assign value to parameter arrays.

```
parameter Real z[:] = {0.1, 0.3, 0.5, 0.7, 0.9};
parameter Real z[:] = 0.1 : 0.2 : 0.9;
```

```
parameter Integer puntos[:] = 2:2:10;
```

```
parameter Real x[2,3] = { {1,2,3} , {4,5,6} };
parameter Real x[2,3] = {  1:3    ,  4:6    };
parameter Real x[2,3] = [ 1, 2, 3 ;  4, 5, 6 ];
```

```
parameter Real y[2,3,4] =
                { { {10,20,30,40},{50,60,70,80},{90,10,11,12} },
                  { {11,21,31,41},{51,61,71,81},{91,11,12,13} }  };
```

Table 2.8: Examples of array constructors with iterators.

Fragment of Modelica code	**Comment to the code**

`parameter Real x[:] =`
`      { i^2`
`        for i in 1:5 };`
	The vector assigned to x is: $\{1, 4, 9, 16, 25\}$

`parameter Real A[:,:] =`
`      { i^2`
`        for i in 1:n,`
`            j in 1:m };`
	The matrix assigned to A is: $\begin{pmatrix} 1 & ... & 1 \\ 4 & ... & 4 \\ ... & ... & ... \\ n^2 & ... & n^2 \end{pmatrix}_{n\times m}$

`parameter Real B[:,:] =`
`      { if i == j then i else 0`
`           for i in 1:n,`
`               j in 1:n };`
	The matrix assigned to B is: $\begin{pmatrix} 1 & 0 & 0 & ... & 0 \\ 0 & 2 & 0 & ... & 0 \\ 0 & 0 & 3 & ... & 0 \\ ... & & & ... & \\ 0 & 0 & 0 & ... & n \end{pmatrix}_{n\times n}$

Table 2.9: Example of *for-clause* in *equation* section.

Equation	Fragment of Modelica code

$y = \sum_{i=0}^{n} c_{i+1} \cdot x^i$	`   parameter Real c[:];`
	`   Real    x, y;`
	`   parameter Integer n = size(c,1) - 1;`
	`   Real a[n+1];`
	`equation`
	`   a[1] = 1;`
	`   for i in 1:n loop`
	`      a[i+1] = a[i] * x;`
	`   end for;`
	`   y = c * a;`

Table 2.10: Examples of array equations and algorithms. Note that the computational causality of equations is calculated by the modeling environment, whereas the computational causality of assignments is explicitly defined by the model developer.

Array equation	For clause in algorithm section

`   Real x[3], y[3];`
`   Real z;`
`equation`
`   y = z * x;`
	`   Real x[3], y[3];`
	`   Real z;`
	`algorithm`
	`   for i in 1:size(x,1) loop`
	`      y[i] := z * x[i];`
	`   end for;`

`   Real x[3], y[3];`
`   Real z;`
`equation`
`   z = x * y;`
	`   Real x[3], y[3];`
	`   Real z;`
	`algorithm`
	`   z := 0;`
	`   for i in 1:size(x,1) loop`
	`      z := z + y[i] * x[i];`
	`   end for;`

`   Real x[3], y[4];`
`   Real A[3,4];`
`equation`
`   x = A * y;`
	`   Real x[3], y[4], A[3,4];`
	`algorithm`
	`   for i in 1:size(A,1) loop`
	`      x[i] := 0;`
	`      for j in 1:size(A,2) loop`
	`         x[i] = x[i] + A[i,j]*y[j];`
	`      end for;`
	`   end for;`

An **algorithm section** is composed of a sequence of assignments that is evaluated by the modeling environment in the same order as written. A variable can be evaluated more than once within an algorithm section. However, a variable evaluated in an algorithm section cannot be evaluated in other algorithm section or equation.

The modeling environment analyzes the computational causality of the complete model, and sorts the equations and algorithm sections attending to the evaluation order of the variables. During the computational causality analysis and the model sorting, each algorithm section is handled as an indivisible block, with explicitly defined computational inputs and outputs.

The treatment of algorithm sections in the causality analysis is as follows. Suppose that n variables are evaluated (appear on the left-hand side of the assignments) from the algorithm section. Then, the algorithm section is replaced by n dummy equations. In each of these dummy equations intervene the n variables evaluated from the algorithm section, and also all the input variables to the algorithm section. This is, the variables that appear only on the right-hand side expressions of the algorithm section assignments. For instance, let's consider the following algorithm section:

```
algorithm
   x1 := g1(y1,...,ym);
   ...
   xn := gn(y1,...,ym);
```

The computational inputs are $y_1, \ldots, y_m$, and the computational outputs are $x_1, \ldots, x_n$. In analyzing the computational causality of the model, this algorithm section is replaced by the following n dummy equations:

$$\begin{aligned} & f_1(x_1, ..., x_n, y_1, ..., y_m) = 0 \\ & ... \\ & f_n(x_1, ..., x_n, y_1, ..., y_m) = 0 \end{aligned} \tag{2.42}$$

As $x_1, \ldots, x_n$ appear in all the equations, these n equations form an algebraic loop (system of simultaneous equations) and will stay together during the model sorting (in the same diagonal block of the BLT matrix, as will be explained in Lesson 4). In the sorted model, this algebraic loop will be placed at a point in which the value of the variables $y_1, \ldots, y_m$ has already been calculated, or can be calculated together with $x_1, \ldots, x_n$. Once the computational causality analysis is finished and the model has been sorted, the set of dummy equations is replaced by the algorithm section.

2.5 Radial heat transfer in a pipe

In this section, we will analyze the stationary heat transfer in the radial direction of an insulated steel pipe. The materials have constant thermal conductivities and there is convection at the boundaries. The system is depicted in Figure 2.12.

The pipe is cylindrical, made of steel, and has a length $L = 1$ m. The steam that circulates inside the pipe is at a constant temperature $T_1 = 418$ K. To reduce the heat loss to the ambient air, the steel pipe is surrounded by an insulation material. The ambient air temperature oscillates between 283 K and 300 K during the day. The physical parameters of the system are listed in Table 2.11.

The thermal equivalent circuit model of the system is shown in the lower part of Figure 2.12. Observe the analogy between the temperature and the voltage; the heat flow and the electric current; and the thermal resistance and the electric resistance. Denoting by q the heat power (units: W) that circulates from the steam to the ambient air, the system model is composed by Eqs. (2.43) – (2.48), and the physical parameters listed in Table 2.11. The temperatures are expressed in Kelvin.

$$
\begin{aligned}
T_1 &= 418 && (2.43)\\
T_1 - T_2 &= q \cdot \frac{1}{h_1 \cdot 2 \cdot \pi \cdot r_1 \cdot L} && (2.44)\\
T_2 - T_3 &= q \cdot \frac{\ln\left(r_2/r_1\right)}{2 \cdot \pi \cdot \kappa_1 \cdot L} && (2.45)\\
T_3 - T_4 &= q \cdot \frac{\ln\left(r_3/r_2\right)}{2 \cdot \pi \cdot \kappa_2 \cdot L} && (2.46)\\
T_4 - T_5 &= q \cdot \frac{1}{h_2 \cdot 2 \cdot \pi \cdot r_3 \cdot L} && (2.47)\\
T_5 &= 291.5 - 8.5 \cdot \sin\left(\frac{2 \cdot \pi \cdot t}{86400}\right) && (2.48)
\end{aligned}
$$

The model can be described as shown in Modelica Code 2.8. The variables that represent temperatures are of the *Modelica.SIunits.Temperature* type, which is declared in the MSL as follows:

```
type Temperature = ThermodynamicTemperature;
type ThermodynamicTemperature = Real (
    final quantity = "ThermodynamicTemperature",
    final unit = "K",
    min = 0.0,
    start = 288.15,
    nominal = 300,
    displayUnit = "degC" );
```

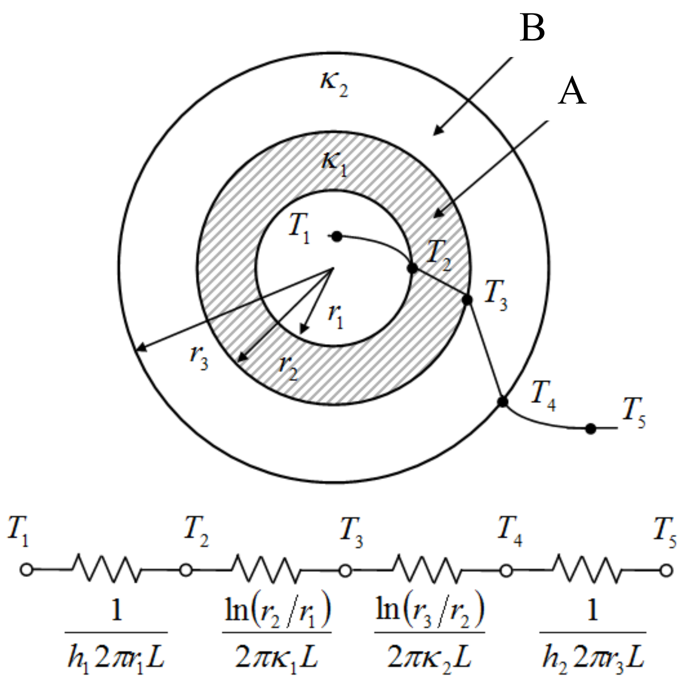

B
A
κ_2
κ_1
T_1
T_2
T_3
T_4
T_5
r_1
r_2
r_3
T_1 T_2 T_3 T_4 T_5
$\frac{1}{h_1 2\pi r_1 L}$
$\frac{\ln(r_2/r_1)}{2\pi\kappa_1 L}$
$\frac{\ln(r_3/r_2)}{2\pi\kappa_2 L}$
$\frac{1}{h_2 2\pi r_3 L}$

Figure 2.12: Temperature profiles for radial heat transfer in an insulated pipe (above) and equivalent thermal circuit (below). There is heat conduction in material A (the steel pipe) and material B (the insulation), and convection on the boundaries (the steam inside the pipe and the surrounding ambient air).

Table 2.11: Physical parameters of the system in Figure 2.12.

Component	**Quantity**	**Value**	**Units**
Steam	Heat transfer coefficient	$h_1 = 11350$	W/(m^2·K)
Steel pipe	Inner radius	$r_1 = 0.025$	m
	Outer radius	$r_2 = 0.035$	m
	Thermal conductivity	$\kappa_1 = 45$	W/(m·K)
Insulation	Outer radius	$r_3 = 0.06$	m
	Thermal conductivity	$\kappa_2 = 0.087$	W/(m·K)
Ambient air	Heat transfer coefficient	$h_2 = 1.32\left(\frac{\lvert T_4 - T_5\rvert}{2\cdot r_3}\right)^{0.25}$	W/(m^2·K)

The `unit` attribute specifies the units employed in the model equations, whereas the `displayUnit` attribute indicates the units that the modeling environment should use when displaying the simulation results of this variable (performing automatically the operations for units conversion). In this case, the temperature is expressed in Kelvins when intervening in the model equations and is displayed in Celsius degrees.

The simulated evolution of the temperatures during 86400 s (one day) is displayed in Figure 2.13. Observe that the plots have different vertical-axis (ordinate) scales: the larger drops in temperature are produced in the insulating layer and the surrounding ambient air.

Let's elaborate on this model a bit more. Suppose now that the air temperature (T_5) is not described by a known function of time, but it has to be interpolated from the temperature readings recorded at specific times in a day. We will program a Modelica function to calculate the temperature at any time of the day by linear interpolation between the available time-temperature recordings. To this end, firstly we will introduce the declaration and use of functions in Modelica.

Certain functions are built into the Modelica language. Some examples of **built-in functions** are `abs` (absolute value function); `sign` (sign function); `sqrt(v)` (square root function); and `div(x,y)` and `rem(x,y)` (quotient and remainder of the division x/y). The arguments of the built-in functions can be scalars or arrays. In the latter case, the function is applied to each of the array components, returning an array. For instance, `sqrt( {1,2,3} )` is equivalent to `{ sqrt(1), sqrt(2), sqrt(3) }`; and `rem( {10,20,30}, {2,3,4} )` is equivalent to `{ rem(10,2), rem(20,3), rem(30,4) }`.

In addition, Modelica provides the **function class** to create user-defined functions. The declaration of a function can consist of:

– The **function interface**, containing the declaration of the function arguments (input variables) and returned values (output variables). The **input** and **output** keywords are used to distinguish between input and output variables.

– The **local variables** employed in the function algorithm, declared within a **protected section**.

– The body of the function, consisting of an **algorithm section** or a **call to an external function** written in a programming language (typically, C or Fortran 77).

User-defined functions must fulfill the following rule: *functions cannot have internal memory.* This means that given the same input, the function always must return the same output.

```
model HeatTransferPipe
  import SI = Modelica.SIunits;
  import Modelica.Constants.pi;
  import Modelica.Math.log;

  parameter SI.Radius r1=0.025 "Inner radius of steel pipe";
  parameter SI.Radius r2=0.035 "Outer radius of steel pipe";
  parameter SI.Radius r3=0.06  "Outer radius of insulation";
  parameter SI.Length L=1 "Pipe length";

  constant SI.ThermalConductivity k1=45 "Thermal conductivity of steel pipe";
  constant SI.ThermalConductivity k2=0.087 "Thermal conductivity of insulation";

  constant SI.CoefficientOfHeatTransfer h1=11350
    "Heat transfer coefficient of steam";
  SI.CoefficientOfHeatTransfer h2 "Heat transfer coefficient of ambient air";

  parameter SI.Temperature T1=418 "Steam temperature";
  SI.Temperature T2 "Temperature of pipe inner surface";
  SI.Temperature T3 "Temperature of pipe outer surface";
  SI.Temperature T4 "Temperature of insulation outer surface";
  SI.Temperature T5 "Temperature of ambient air";

  SI.HeatFlowRate q "Heat power from the steam to the ambient air";
equation
  h2 = 1.32*(abs(T4 - T5)/(2*r3))^0.25;
  T1 - T2 = q/(h1*2*pi*r1*L);
  T2 - T3 = q*log(r2/r1)/(2*pi*k1*L);
  T3 - T4 = q*log(r3/r2)/(2*pi*k2*L);
  T4 - T5 = q/(h2*2*pi*r3*L);
  T5 = 291.5 - 8.5*sin(2*pi*time/86400);
end HeatTransferPipe;
```

Modelica Code 2.8: Radial heat transfer in an insulated steel pipe.

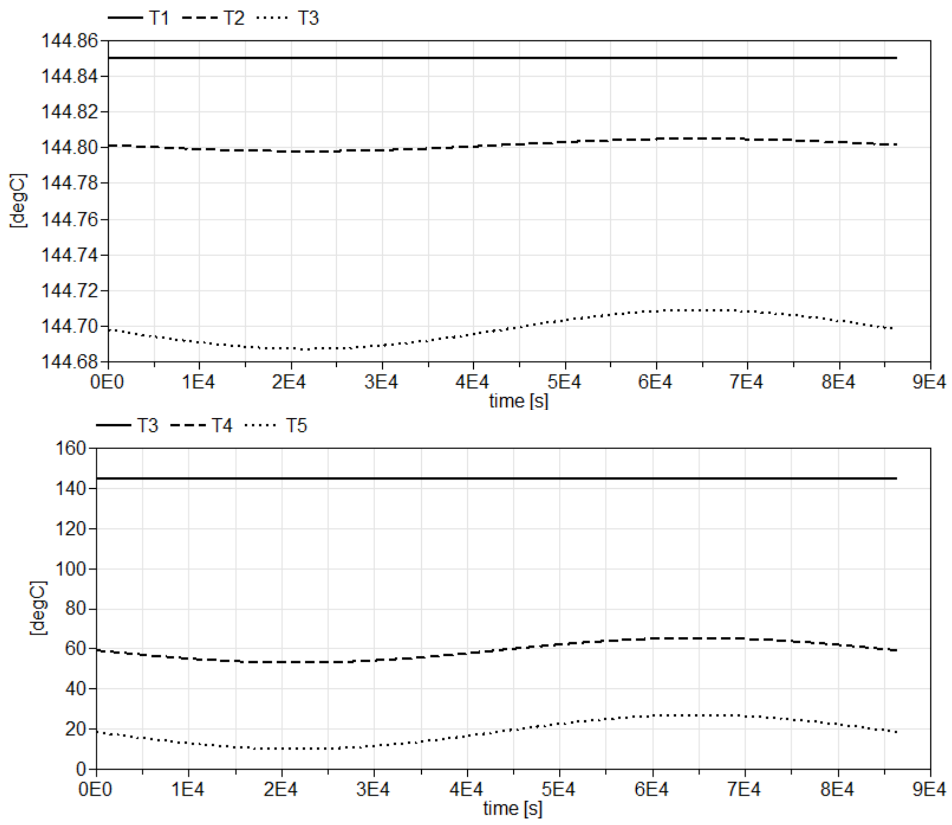


Figure 2.13: Result obtained of simulating the Modelica Code 2.8 during 86400 s.

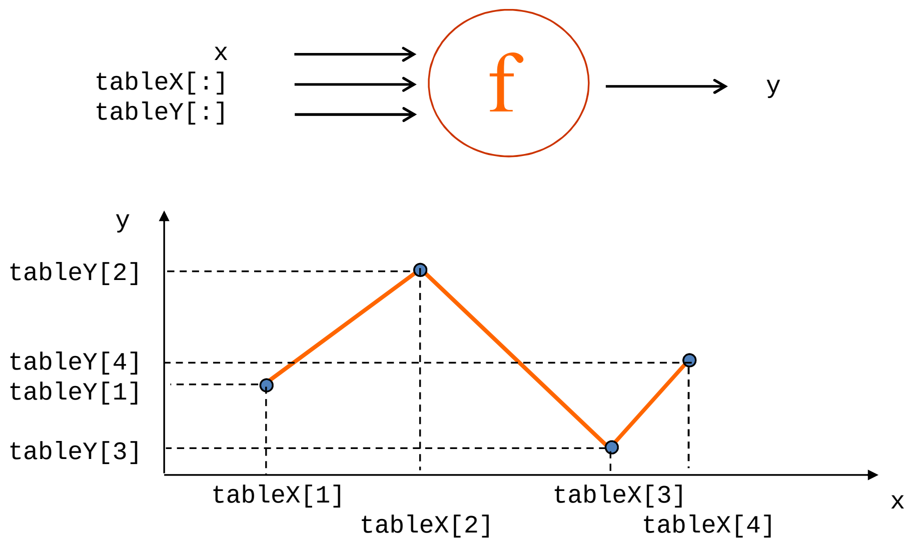


Figure 2.14: Function interface (above) and example of linear interpolation (below).

Let's return to the problem of programming a function that performs linear interpolation. The function interface is depicted in Figure 2.14. The function computes an output value (`y`) by interpolating the input value (`x`) against a set of data points (`tableX` and `tableY` vectors). The function definition and a fragment of the heat transfer model are shown in Modelica Code 2.9. The air temperature calculated by interpolation is shown in Figure 2.15.

The **assert** sentence is employed in the `LinearInterpolation` function (see Modelica Code 2.9). This sentence allows to specify conditions that must be satisfied during the simulation run. Its syntax is: `assert( logical_expression, error_message );` The logical expression should be *true* at any time. If it becomes *false*, the simulation run is aborted and the error message is shown in the log window.

The `LinearInterpolation` function returns one value. The definition of functions that returns more than one value is illustrated by the following example:

```
function Polar2Cartesian
// Convert from Polar Coordinates to Cartesian Coordinates
   input  Real ang, r;
   output Real x, y;
algorithm
   x := r * cos(ang);
   y := r * sin(ang);
end Polar2Cartesian;
```

This function can be called as follows:

```
(x1,y1) = Polar2Cartesian(ang1, rad1);
```

```
function LinearInterpolation
  input Real x "Independent variable";
  input Real tableX[:] "Independent variable points";
  input Real tableY[:] "Dependent variable points";
  output Real y "Interpolated value";
protected
  Integer n;
  Real slope;
algorithm
  n := size(tableX, 1);
  assert(size(tableX, 1) == size(tableY, 1),
    "Error: tableX and tableY with different size");
  assert(x >= tableX[1] and x <= tableX[n],
    "Error: independent variable is out of range");
  for i in 1:n - 1 loop
    if x >= tableX[i] and x <= tableX[i + 1] then
      slope := (tableY[i + 1] - tableY[i])/(tableX[i + 1] - tableX[i]);
      y := tableY[i] + slope*(x - tableX[i]);
    end if;
  end for;
end LinearInterpolation;


model HeatTransferPipeInterp

  parameter SI.Time valTime[:]=0:3600:86400;
  parameter SI.Temperature valTemp[:]={291.5,293.0,295.5,300.0,302.5,305.5,
      306.0,306.5,308.5,310.0,312.5,313.5,313.5,312.5,310.0,309.0,305.5,300.5,
      298.5,293.5,290.5,286.0,280.5,275.5,273.0};

  ...
equation
  ...
  // T5 = 291.5 - 8.5*sin(2*pi*time/86400);
  T5 = LinearInterpolation(x=time, tableX=valTime, tableY=valTemp);
end HeatTransferPipeInterp;
```

Modelica Code 2.9: Linear interpolation funcion and its invocation.

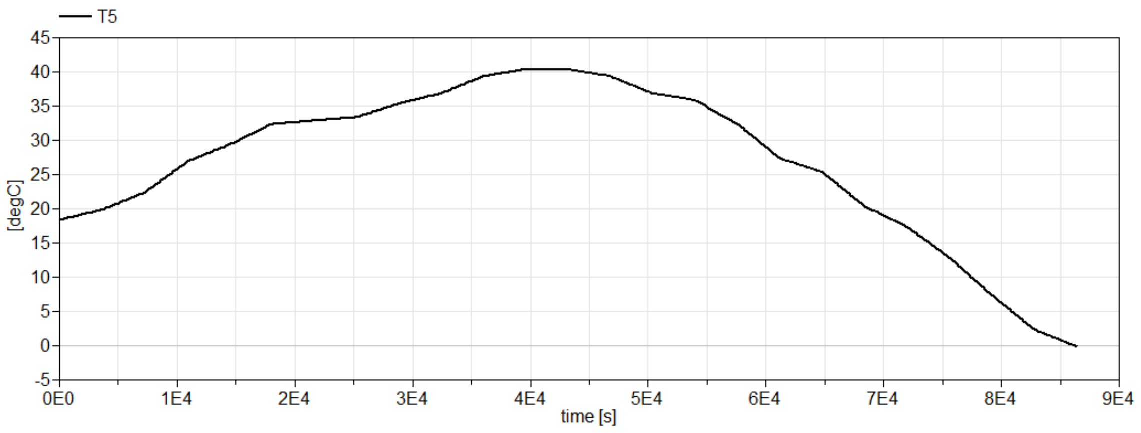


Figure 2.15: Air temperature calculated by linear interpolation.

By-default values of the inputs can be specified in the function declaration. These values will be employed if the inputs are not present in the function call. For instance, given this declaration

```
function Polar2Cartesian
   input  Real ang = 0, r = 1;
   output Real x, y;
algorithm
   x := r * cos(ang);
   y := r * sin(ang);
end Polar2Cartesian;
```

the following calls are equivalent:

```
(x1,y1) = Polar2Cartesian(0, 1);
(x1,y1) = Polar2Cartesian();
(x1,y1) = Polar2Cartesian(0);
(x1,y1) = Polar2Cartesian(r=1);
```

2.6 Further reading

The description of continuous-time atomic models in Modelica was introduced in this lesson, using a series of examples. Additional examples, and the description of other Modelica features, can be found in (ModelicaTM 2000), (Otter 2009), (Fritzson 2011) and (Tiller 2001). These books provide a didactic introduction to Modelica. Readers interested in a deeper description of the Modelica language can find more complete information in (Fritzson 2015). On the other hand, the most recent language specification is available on the Modelica Association website (ModelicaWebSite 2017). The use of these references is recommended to complement the explanations provided in Lessons 2, 3 and 7.

Problem solving by modeling and simulation in the Chemical Engineering domain is introduced in (Cutlip & Shacham 1999). The models of the radial heat transfer in a pipe and the heat conduction in a wall (cf. Sections 2.5 and 9.7) have been extracted from this excellent book.

Lesson 3

Model libraries

3.1 Introduction
3.2 Electrical library
3.3 Longitudinal vibrations of a bar
3.4 Longitudinal heat conduction in a bar
3.5 Control of level and temperature in a tank
3.6 Dissipation of heat generated in a circuit
3.7 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Design model libraries applying the object-oriented modeling methodology.

– Define connectors, components and compound models in Modelica.

– Use class inheritance in Modelica.

– Use replaceable classes in Modelica.

– Describe in Modelica models with a regular structure, using both arrays of components and arrays of variables.

– Define and use record classes in Modelica.

– Use the if-clause and for-clause of Modelica.

– Use the inner/outer construct of Modelica.

– Develop and use libraries of continuous-time models in Modelica.

3.1 Introduction

The design and implementation of Modelica libraries are discussed, as in the previous lesson, through a series of examples.

The rectifier circuit shown in Figure 2.1 is modeled in Section 3.2 from a different perspective. Instead of describing the circuit as an atomic model, a Modelica library of electrical components is developed, and the circuit is composed instantiating and connecting model classes from the library. This example serves to illustrate the declaration of connectors, the modeling of component interfaces, the definition of compound models by instantiating and connecting model classes, the definition of derived classes, the definition of Modelica libraries, the access to library components, and alternative ways of saving libraries to disk. The example of the electrical library also allows to introduce the concept of replaceable classes.

A model to analyze the longitudinal vibrations of a bar is described in Section 3.3. This example allows to introduce the use of record classes and illustrates the use of for-loops for describing equations involving arrays of variables.

The example in Section 3.4 is intended to introduce the declaration of vectors of components and the use of for-loops for describing the connection of their connectors. The example consists in modeling the longitudinal heat conduction in a bar. The model is described in two different ways. Firstly, as an atomic model, using vectors of variables and for-loops containing equations. Secondly, a Modelica library is implemented, and the bar model is composed from it, declaring vectors of components and writing the connection sentences within for-loops.

The control of level and temperature in a tank is addressed in Section 3.5. Firstly, an atomic model of the system is developed. Next, the system is decomposed into parts, and a model library is designed and implemented. This example tries to illustrate the application of the object-oriented methodology and its implementation in Modelica.

Finally, the inner/outer construct of Modelica is explained in Section 3.6. The example consists in modeling the dissipation of heat generated by an electric circuit.

3.2 Electrical library

Let's revisit the modeling of the rectifier circuit shown in Figure 2.1. Instead of describing the circuit as an atomic model, now we will apply the object-oriented modeling methodology. To this end, we are going to decompose the circuit into its components, analyze how to model the interaction among these components, program model classes describing the components, arrange the classes into a model library, and finally use the library to compose the circuit model.

The rectified circuit is composed of four different classes of electric component: voltage source, diode, resistor and capacitor. Their symbols and constitutive relationships were displayed in Figure 2.2. The components are connected by connecting their electric pins. The pins connected at a connection point (circuit node) have the same voltage and the sum of all the currents is zero. The **connector** class of Modelica can be used for modeling the electric pin as shown below.

```
connector Pin
   Modelica.SIunits.Voltage        u;
   flow Modelica.SIunits.Current   i;
end Pin;
```

The class definition starts with the **connector** keyword followed by the class name (`Pin`), and ends with the **end** keyword followed by the class name and a semicolon. The class body contains the declaration of the connector's variables. The **flow** keyword allows to distinguish between across and through variables. It is written preceding the declaration of the through variables, as is the electric current in this case.

The component **interface** is composed of the variables that describe the component interaction with the rest of the system, and the parameters. A model class describes a type of system. By setting the value of the model class parameters, an element in particular (a particular instance) of the model class is specified.

When developing model libraries, it is often a good practice to define the components' interfaces separately. This facilitates the reuse of the interface definitions and makes easier to tell whether two model classes have the same interface. In this way, it can be reasoned that two model classes have the same interface if they have a superclass in common: the interface definition. The interest in knowing whether two classes have the same interface comes from the fact that having the same interface is a necessary condition (but not sufficient) for a component of a class to be replaceable by a component of the other class.

The rectifier circuit components, this is, the voltage source, diode, resistor and capacitor, are two-pin components. The two pins can be declared in a class that will be inherited by the four component classes. On the other hand, the physical quantity describing the voltage drop between the pins intervenes in the constitutive relationships of the four components. The two pins and the voltage drop are declared in the following class.

```
partial model TwoPins
   Pin   p, n;
protected
   Modelica.SIunits.Voltage u   "Voltage drop (= p.u - n.u)";
equation
   u = p.u - n.u;
end TwoPins;
```

Observe that the dot notation is used to reference the variables of the connectors (e.g., `p.u` is the `u` variable of the `p` connector). The **partial** keyword indicates that `TwoPins` is a *partial class*, this is, a class that describes the system partially. As a partial class does not completely describe the system behavior, it is intended to be inherited, not instantiated.

The **protected** keyword indicates the beginning of a section in which the declared variables, inherited classes and instantiated components are protected elements. This means that is not possible to access them from outside the class using dot notation. By-default, it is assumed that the elements are public. The `u` variable has been declared as protected, and the `p` and `n` connectors are public. The **public** keyword allows to explicitly signal the beginning of a section where public components are declared. An arbitrary number of public and protected sections can be declared in a class.

Two-pin components such as resistor, capacitor, induction and diode, have the following property: the current entering a pin is equal to the current leaving the other pin. In addition, the sign convention for passive components says that the current arrow points into the positive voltage terminal of the element. The following class inherits `TwoPins` and describes the additional behavior.

```
partial model OnePort
   extends TwoPins;
protected
   Modelica.SIunits.Current  i "Current entering p and leaving n";
equation
   i = p.i;
   i = -n.i;
end OnePort;
```

Inheritance is declared writing the **extends** keyword followed by the name of the superclass, and optionally by the assignation of values to the superclass parameters. As Modelica supports **multiple inheritance**, a class can contain an arbitrary number of extends clauses.

The classes that describe the resistor, diode and capacitor are subclasses of `OnePort`. They are defined below.

```
model Resistor "Ideal resistor"
   extends OnePort;
   parameter Modelica.SIunits.Resistance R "Resistance";
equation
   u = R*i;
end Resistor;

model Diode "Ideal diode"
   extends OnePort;
   parameter Modelica.SIunits.Current Is = 1e-9
                   "Saturation current";
   parameter Modelica.SIunits.Voltage Vt = 0.025 "Thermal voltage";
equation
   i = Is * ( exp(u / Vt) - 1);;
end Diode;

model Capacitor "Ideal capacitor"
   extends OnePort;
   parameter Modelica.SIunits.Capacitance C "Capacitance";
equation
   C*der(u) = i;
end Capacitor;
```

Observe that we have not assigned a value to the parameters of the `Resistor` and `Capacitor` classes (the `R` and `C` parameters). In this situation, the **by-default value** is zero if the parameter is of Real or Integer type, and *false* if Boolean. Parameter values can be set and modified when declaring instances of the classes or inheriting them, and at the experiment definition.

The sign convention for the voltage source is the opposite of the sign convention for passive components. For power sources (active components), the current arrow points outward the positive voltage terminal (see Figure 2.2). Let's modify the `OnePort` class as shown below, so that the sign convention depends on the value assigned to the `active` Boolean parameter. On the other hand, remember the sign convention for the through variables of connectors: the through variable is positive while flows into the component.

```
partial model OnePort
   extends TwoPins;
   parameter Boolean active = false;
protected
   Modelica.SIunits.Current  i "Current through the component";
equation
   if ( active ) then
```

```
      i = n.i;
   else
      i = p.i;
   end if;
   p.i = - n.i;
end OnePort;
```

The **if-clause** allows to specify conditional equations. This is, equations that form part of the model only while certain condition is satisfied. If the `active` parameter is *true*, the current through the component is equal to the current at the negative terminal (`i = n.i;`). If `active` is *false*, the current through the component is equal to the current at the positive terminal (`i = p.i;`).

In general, the **logical condition** of an if-clause is a Boolean expression that may be dependent or independent of time. The latter condition is equivalent to state that the if-clause condition is a Boolean expression that depends only on constants and parameters. The numerical treatment of the if-clause depends on it, as described below.

– If the value of the if-clause condition *can change during the simulation run*, then it must be monitored during the simulation run. The switching of the if-clause can be considered an event. In this case, an event condition (see Section 1.4) is associated to the change in the value of the if-clause condition. It is also possible to make a treatment of the if-clause that is not based on events. This will be explained in Lesson 8.

– If the value of the if-clause condition *cannot change during the simulation run*, then it is possible to calculate it before starting the dynamical solution of the model, replacing the if-clause by the equations of its enabled branch. Modeling environments typically perform this substitution during the model translation process, before analyzing the computational causality of the model, and generating the sorted and solved model. As a consequence, the values of the parameters that intervene in time-independent if-clause conditions can be changed when the components are inherited or instantiated, but cannot be changed at the experiment definition.

The voltage source can be defined as shown below. Observe that the `OnePort` class is inherited modifying the value of its `active` parameter.

```
model VsourceAC "Sinusoidal voltage source"
   extends OnePort ( active = true );
   parameter Modelica.SIunits.Voltage   U0;
```

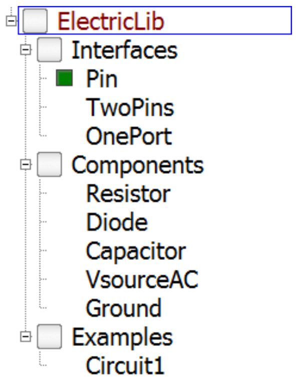


Figure 3.1: Architecture of the electrical library.

```
   parameter Modelica.SIunits.Frequency frec;
   parameter Modelica.SIunits.Angle     phi;
protected
   parameter Modelica.SIunits.AngularFrequency w =
                                 2*Modelica.Constants.pi*frec;
equation
   u = U0 * sin( w * time + phi);
end VsourceAC;
```

The `Ground` component is declared below. By connecting this component to a node circuit, the node is selected as the reference node for the voltage.

```
model Ground "Voltage reference"
   Pin p;
equation
   p.u = 0;
end Ground;
```

Modelica provides the **package** class to facilitate structuring the definition of types, connectors, models, etc. into libraries. Packages can be declared inside other packages, allowing to define hierarchically structured libraries.

Let's arrange the classes of the electrical components into a model library. The complete content of the library will be defined within a package named `ElectricLib`. Within this package, the following three packages are defined (see Figure 3.1): `Interfaces` contains the component interfaces, `Components` the electric components, and `Examples` the rectifier circuit model. The library is defined in Modelica Code 3.1–3.3).

Dot notation allows to reference classes defined within packages. For instance, the `Resistor` class, defined within the `Components` package of the electrical library (see Figure 3.1), can be accessed from outside the electrical library specifying the

```
encapsulated package ElectricLib

  import SI = Modelica.SIunits;
  import Modelica.Constants;

  package Interfaces
    connector Pin
      SI.Voltage u;
      flow SI.Current i;
    end Pin;

    partial model TwoPins
      Pin p, n;
    protected
      SI.Voltage u "Voltage drop (= p.u - n.u)";
    equation
      u = p.u - n.u;
    end TwoPins;

    partial model OnePort
      extends TwoPins;
      parameter Boolean active = false;
    protected
      SI.Current i "Current through the component";
    equation
      if ( active ) then
        i = n.i;
      else
        i = p.i;
      end if;
      p.i = -n.i;
    end OnePort;

  end Interfaces;
```

Modelica Code 3.1: Electrical library (1/3).

```
package Components

  model Resistor "Ideal resistor"
    extends Interfaces.OnePort;
    parameter SI.Resistance R "Resistance";
  equation
    u = R*i;
  end Resistor;

  model Diode "Ideal diode"
    extends Interfaces.OnePort;
    parameter SI.Current Is=1e-9 "Saturation current";
    parameter SI.Voltage Vt=0.025 "Thermal voltage";
  equation
    i = Is*(exp(u/Vt) - 1);
  end Diode;

  model Capacitor "Ideal capacitor"
    extends Interfaces.OnePort;
    parameter SI.Capacitance C "Capacitance";
  equation
    C*der(u) = i;
  end Capacitor;

  model VsourceAC "AC  voltage source"
    extends Interfaces.OnePort( active=true );
    parameter SI.Voltage U0;
    parameter SI.Frequency frec;
    parameter SI.Angle phi;
  protected
    parameter SI.AngularFrequency w=2*Constants.pi*frec;
  equation
    u = U0*sin(w*time + phi);
  end VsourceAC;

  model Ground "Voltage reference"
    Interfaces.Pin p;
  equation
    p.u = 0;
  end Ground;

end Components;
```

Modelica Code 3.2: Electrical library (2/3).

```
  package Examples

    model Circuit1
      // Voltage source
      parameter SI.Voltage U0=5;
      parameter SI.Frequency frec=100;
      parameter SI.Angle phi=0;
      // Resistors
      parameter SI.Resistance R1=100;
      parameter SI.Resistance R2=100;
      // Capacitor
      parameter SI.Capacitance C=1e-6;
      // Diode
      parameter SI.Current Is=1e-9;
      parameter SI.Voltage Vt=0.025;

      // Components
      Components.Resistor Resist1(R=R1);
      Components.Resistor Resist2(R=R2);
      Components.Capacitor Cond(C=C);
      Components.Diode Diode(Is=Is, Vt=Vt);
      Components.VsourceAC VS(U0=U0, frec=frec, phi=phi);
      Components.Ground ground;
    equation
      connect(VS.p, Resist1.p);
      connect(Resist1.n, Diode.p);
      connect(Resist1.n, Resist2.p);
      connect(Resist1.n, Cond.p);
      connect(VS.n, ground.p);
      connect(Diode.n, ground.p);
      connect(Resist2.n, ground.p);
      connect(Cond.n, ground.p);
    end Circuit1;

  end Examples;

end ElectricLib;
```

Modelica Code 3.3: Electrical library (3/3).

complete path: `ElectricLib.Components.Resistor`. The name in dot notation consists of a sequence of dot-separated identifiers. The last identifier is the class name. The identifiers before the last one designate the path in the library hierarchical structure to reach the class.

Class names in dot notation represent relative paths. The starting point of the relative path is described by the first identifier of the name. The search of the class designated by the first identifier is firstly conducted at the hierarchical level where the referencing class is. If not found, the search continues ascending through the hierarchical structure.

Therefore, if referencing and referenced classes are in same library, it is not necessary to write the complete path of the referenced class. As shown in Modelica Code 3.3, the `Resistor` class can be referenced from the `Circuit1` class simply by writing `Components.Resistor`.

Observe in Modelica Code 3.1 that the electrical library has been declared as an **encapsulated package**. The concept of encapsulating a package is intended to facilitate its portability and the class maintenance. From within an encapsulated package, it is not possible to reference directly classes located outside the package. The objective is to minimize the number of changes required when adapting the package to changes in the class hierarchy. Modelica provides the **import sentence** to allow accessing classes external to the encapsulated package. When a class is imported, it becomes "visible" from within the encapsulated package.

The following two sentences, extracted from Modelica Code 3.1, allow to illustrate two different ways of using the import sentence:

```
import SI = Modelica.SIunits;
import Modelica.Constants;
```

The first one makes the `Modelica.SIunits` class visible from within `ElectricLib` and also declares `SI` as an abbreviation of `Modelica.SIunits`. Abbreviations such as `SI.Voltage` and `SI.Current` can be used to reference `Modelica.SIunits.Voltage` and `Modelica.SIunits.Current` respectively.

The second sentence makes visible the `Modelica.Constants` class from within `ElectricLib`. This implies that `Modelica.Constants` can be the starting point of relative paths. For instance, the `Constants.pi` constant is employed in the definition of the `VsourceAC` class (see Modelica Code 3.2).

The import sentence can be used in a third way, as illustrated by the following line of code.

```
import Modelica.Math.*;
```

This import sentence makes visible all the classes declared within `Modelica.Math` and allows to reference them directly. For instance, the `sin` function declared within the `Modelica.Math` package can be referenced as `sin`.

In any case, in order to avoid hidden dependencies, the **import sentences are not inherited**.

It can be observed in Modelica Code 3.3 that the connection of the component connectors is described using **connect** sentences. As was explained, these sentences are translated into equations by the modeling environment. It is worthy of mention that if a **connector is left unconnected**, the modeling environment assumes that its through variables are zero, *adding automatically the corresponding equations to the model.*

It can also be observed in Modelica Code 3.3 how the parameter values are set when the components are instantiated. If a component is itself a composed class, the dot notation can be employed for **referencing the parameters**. This is illustrated in the following example.

```
model Resistor
   parameter Modelica.SIunits.Resistance R;
   ...
end Resistor;

model SubCircuit
   Resistor R1( R=1 ), R2( R=10 );
   ...
end SubCircuit;

model Circuit
   SubCircuit SC ( R1.R=3, R2.R=30 );
   ...
end Circuit;
```

Concerning how **packages are saved to file**, there are basically the two following options.

– *Save the entire package to a single file.* This has typically the name of the package and the **.mo** extension. In this way, the `ElectricLib` package would be saved to a file named *ElectricLib.mo.*

– *Save the package in several files*, structured in a directory hierarchy that mimics the package structure. The procedure is as follows.

1. Create a directory for each package, maintaining the same hierarchical structure as the packages. Give to each directory the same name as the corresponding package.
2. Create a file named **package.mo** within each directory. Write to this file the package declaration, but excluding the classes declared within the package.
3. Declare each class in a separate file, saved into the corresponding directory. The file name has to be the same as the class name, with *.mo* extension. The first sentence in each of these files must be a **within** sentence specifying the class path in the package hierarchy. For instance, the first sentence in the *Resistor.mo* file should be:

   ```
   within ElectricLib.Components;
   ```

The **MODELICAPATH** environment variable of the operative system allows the user to specify the directories (as a list separated by semicolons) where the modeling environment has to search for packages.

Returning to our discussion on the electrical library, observe that the parameters of the library models are time-independent variables, whose value can be modified at instantiation and inheritance, and at the experiment definition. Modelica not only supports the parametrization of time-independent variables. It also supports the declaration of **replaceable classes**. The following example is used to illustrate this feature.

Suppose that we have developed two different models of an electric resistor. The first one is the `Resistor` class shown in Modelica Code 3.2. In the second model, which is defined below, the effect of temperature on the resistance is considered.

```
model ResistorTemp "Resistor with temperature dependence"
   extends Interfaces.OnePort;
   parameter SI.Resistance R = 1 "Resistance at reference temperature";
   parameter Real alpha(unit="K-1") = 5e-05 "Temperature coefficient";
   parameter SI.Temperature Tref  = 298.15 "Reference temperature";
   SI.Temperature Temp "Resistor temperature";
equation
   u = R*( 1 + alpha*(Temp-Tref) )*i;
end ResistorTemp;
```

Depending on the study objective, we will need to use `Resistor` or `ResistorTemp` to describe the circuit's resistors. Modelica allows to declare the object classes as replaceable, and redeclare them. For instance, the `R1` and `R2` resistors are declared as **replaceable** components in the model shown below.

```
model Circuit
   replaceable Resistor R1(R=100), R2(R=200);
   Resistor R3(R=300), R4(final R=400);
   ...
end Circuit;
```

The class of `R1` and `R2` can be redeclared when the `Circuit` class is instantiated and inherited. For instance, `R1` and `R2` are redeclared as objects of the `ResistorTemp` class in the `CircuitT` model defined below.

```
model CircuitT
   extends Circuit (
                      redeclare ResistorTemp R1(alpha=1e-4),
                      redeclare ResistorTemp R2 );
end CircuitT;
```

When declaring an object of replaceable class, Modelica allows to impose a condition on the replacing class: to have a specific superclass, which typically is the class where the component interface is defined. For instance, it can be specified that the classes of the `R1` and `R2` components must be derived classes of the `OnePort` class.

```
model Circuit
   replaceable Resistor R1(R=100) extends Interfaces.OnePort;
   replaceable Resistor R2(R=200) extends Interfaces.OnePort;
   Resistor R3(R=300), R4(final R=400);
   ...
end Circuit;
```

Modelica allows to redeclare the class of several objects without naming them one by one. Consider the following example.

```
model CircuitA
   replaceable model Device = Resistor;
   Device R1, R2;
   Resistor R3(R=300), R4(final R=400);
   ...
end CircuitA;
```

A replaceable model named `Device` is defined within `CircuitA`. As stated in its declaration sentence, `Device` is equal to the `Resistor` class. The `R1` and `R2` components are objects of the `Device` class. If not explicitly stated otherwise, `R1` and `R2` components are of the `Resistor` class. However, as `Device` has been declared as replaceable, it can be redeclared when `CircuitA` is instantiated or inherited. For instance, redeclaring `Device` as equal to the `Capacitor` class (see the declaration below), `R1` and `R2` are capacitors in `CircuitB`.

```
model CircuitB
   extends CircuitA ( redeclare model Device = Capacitor );
end CircuitB;
```

In addition to replaceable models, Modelica allows to declare replaceable connectors, types, records, blocks, functions and packages. The syntax is analogous, replacing the *model* keyword after *replaceable* and *redeclare* by the corresponding keyword: *connector*, *type*, etc. The declaration and redeclaration of a replaceable type is illustrated in the following example.

Consider the definition of a model that describes an exponential waveform. This model will be used to describe two different generators: a voltage source and a current source. As the units of the variable that describes the waveform can be Volts or Amperes, the type of the variable is declared as of **replaceable type**. It is shown in the following fragment of code.

```
partial model EXP
   replaceable type SignalType = Real;
   parameter SignalType    S1;
   parameter SignalType    S2;
   parameter Modelica.SIunits.Time  TD1(min=0);
   ...
protected
   parameter SignalType    TRANS_INITIAL = S1;
   SignalType              signal;
   Modelica.SIunits.Time   timeStartTran;
equation
   ...
end EXP;
```

The exponential waveform can be used in the definition of voltage and current sources, as shown below.

```
model Vsource
  extends EXP ( redeclare type SignalType=Modelica.SIunits.Voltage );
  ...
end Vsource;

model Isource
   extends EXP ( redeclare type SignalType=Modelica.SIunits.Current );
   ...
end Isource;
```

3.3 Longitudinal vibrations of a bar

Consider a uniform bar of length L, cross section A, mass M and Young's modulus E. The left end of the bar is fixed and the right end is free. An external force $F(t)$ is applied at the free end in the longitudinal direction. The system is depicted in Figure 3.2, where $\xi\,(x,t)$ represents the longitudinal displacement from its equilibrium position at a distance x from the left end and at time t.

The objective is to analyze the resulting vibration of the bar in the axial (longitudinal) direction. The model is developed by dividing the bar into n elements of equal length $\Delta x = L/n$, each of mass $\Delta m = M/n$. The linear momentum of the i-th element is:

$$p_i = \Delta m \cdot \frac{d\xi_i}{dt} \qquad \text{with } i = 1, \ldots, n \tag{3.1}$$

The stress on each element is modeled as the force exerted by strings connected to the element, as shown is Figure 3.3. Therefore, the bar is described as the connection of n masses and n strings. Applying the Newton's law to each element, it is obtained:

$$\frac{dp_i}{dt} = F_{i+1} - F_i \qquad \text{with } i = 1, \ldots, n-1 \tag{3.2}$$

$$\frac{dp_n}{dt} = F - F_n \tag{3.3}$$

The force exerted by the strings can be calculated using Eqs. (3.4) and (3.5), and k can be calculated from Eq. (3.6).

$$F_1 = k \cdot \xi_1 \tag{3.4}$$

$$F_i = k \cdot (\xi_i - \xi_{i-1}) \qquad \text{with } i = 2, \ldots, n \tag{3.5}$$

$$k = \frac{E \cdot A}{\Delta x} \tag{3.6}$$

The bar model is composed of Eqs. (3.1) – (3.6), and the parameter values shown in Table 3.1. Initially, the external force is zero and the bar is at equilibrium. At time equals 0.001 s, the external force changes abruptly to 2000 N, and this value is maintained constant during all the simulation time. The bar model and the experimental setup (`Test` model) are shown in Modelica Code 3.4. The displacements

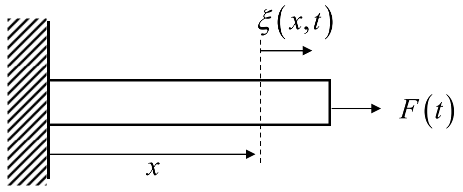
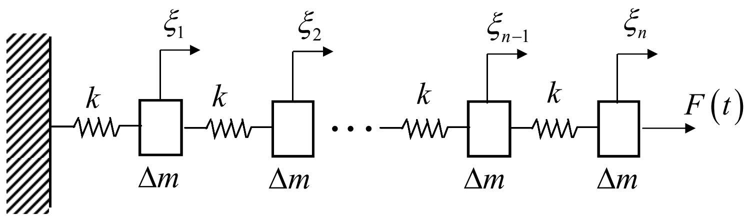


Figure 3.2: External force $F(t)$ applied at the free end of a uniform bar.



$$\Delta m = \rho \cdot A \cdot \Delta x$$

$$k = \frac{\mathrm{E} \cdot A}{\Delta x}$$

Figure 3.3: Model to analyze longitudinal vibrations of a uniform bar.

Table 3.1: Physical parameters of the bar.

Parameter	**Value**	**Units**
Cross section (A)	$8.636E-005$	m^2
Length (L)	1.0	m
Mass (M)	0.233172	kg
Young's modulus (E)	$6.9E+10$	Pa

of the elements $i = 10$, 30 and 100 obtained simulating the `Test` model during 0.006 s are shown in Figure 3.4.

Vector variables have been employed in the `LVib_UnifBar` model to describe the element displacements and linear momenta, and spring forces. For-clauses have been employed to describe the equations.

Modelica provides the **record** class to facilitate grouping the declaration of related parameters and setting their values. A record class cannot contain equations or protected sections.

The separation between model equations and parameter values is a guiding principle in other simulators. An example is the SPICE circuit simulator. The mathematical equations that describe the supported devices (resistors, capacitors, inductors, switches, voltage and current sources, transmission lines, diodes, BJT, JFETs, MOSFETS, MESFETs, etc.) are built into the SPICE simulator. Each of

```
record BarData
  import SI = Modelica.SIunits;
  parameter SI.Area     section;
  parameter SI.Length   length;
  parameter SI.Mass     mass;
  parameter SI.Pressure Young;
end BarData;


package BarCatalog

  record Bar21 = BarData (
      section = 8.636e-005,
      length  = 1,
      mass    = 0.233172,
      Young   = 6.9e10 );

end BarCatalog;


partial model LVib_UnifBar
  import SI = Modelica.SIunits;
  constant Integer n = 100 "Number of elements";
  BarData dBar;
  parameter SI.Length  delta_x = dBar.length/n;
  parameter SI.Mass    delta_m = dBar.mass/n;
  parameter SI.TranslationalSpringConstant k = dBar.Young*dBar.section/delta_x;
  SI.Position eps[n] "Longitudinal displacement";
  SI.Force    F[n]   "Force of springs";
  SI.Momentum p[n]   "Linear momentum";
  SI.Force    Fext    "External force";
equation
  // Linear momentum of elements
  for i in 1:n loop
    p[i] = delta_m*der(eps[i]);
  end for;
  // Newton's law on elements
  for i in 1:(n - 1) loop
    der(p[i]) = F[i + 1] - F[i];
  end for;
  der(p[n]) = Fext - F[n];
  // Force exerted by springs
  F[1] = k*eps[1];
  for i in 2:n loop
    F[i] = k*(eps[i] - eps[i - 1]);
  end for;
end LVib_UnifBar;


model Test
  extends LVib_UnifBar( dBar = BarCatalog.Bar21() );
equation
  // External force in the longitudinal direction
  Fext = if time < 1e-3 then 0 else 2e3;
end Test;
```

Modelica Code 3.4: Longitudinal vibration of a uniform bar.

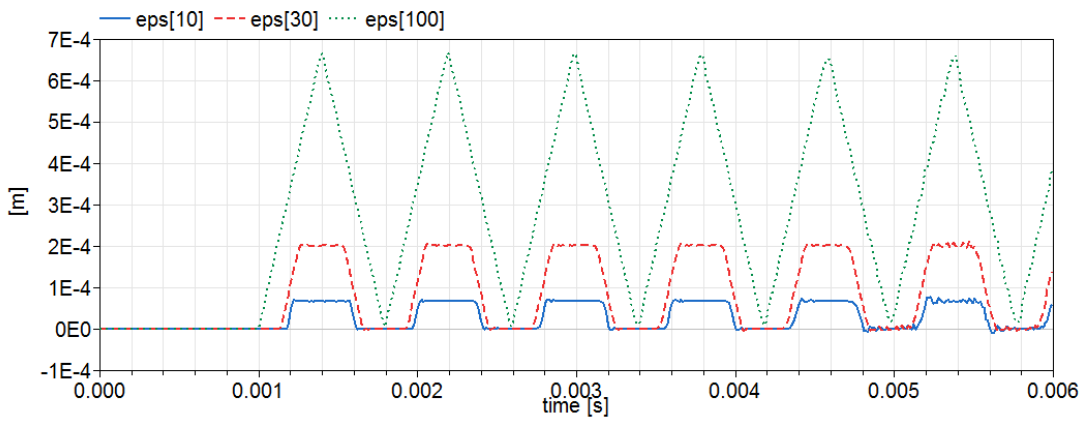


Figure 3.4: Displacement of the elements $i = 10$, 30 and 100 obtained simulating the Test model shown in Modelica Code 3.4 during 0.006 s.

these models contains a set of parameters. Specifying the value of these parameters, the generic behavior described by the device model is particularized for describing the behavior of a particular device. The values of the SPICE parameters are typically provided by the device manufacturers, grouped into model parameter libraries.

The Modelica's **record** class provides support to this principle. An example of use can be found in Modelica Code 3.4. The `BarData` record contains the declaration of the bar parameters. An instance of `BarData`, named `dBar`, has been declared within the `LVib_UnifBar` partial model. The content of `dBar` is accessed using dot notation (e.g., `dBar.section`).

The sets of parameters that describe different bar types can be arranged into a package. The `BarCatalog` package is declared in Modelica Code 3.4 for this purpose. The declaration of the `Bar21` class contains the parameter values of the particular bar that is going to be analyzed in this simulation study. Other records describing other bar types can be included within `BarCatalog`.

Observe the `Test` class defined in Modelica Code 3.4. `Test` inherits `LVib_UnifBar`, and `BarCatalog.Bar21` is assigned to `dBar`. The external force exerted on the bar is also described.

The values declared in `BarCatalog.Bar21` could have been modified when assigned to `dBar`. For instance, declaring

```
model Test
   extends LVib_UnifBar( dBar = BarCatalog.Bar21(length=2) );
   ...
```

the bar length takes the value 2 m, while the other parameters take the values specified in `BarCatalog.Bar21`.

3.4 Longitudinal heat conduction in a bar

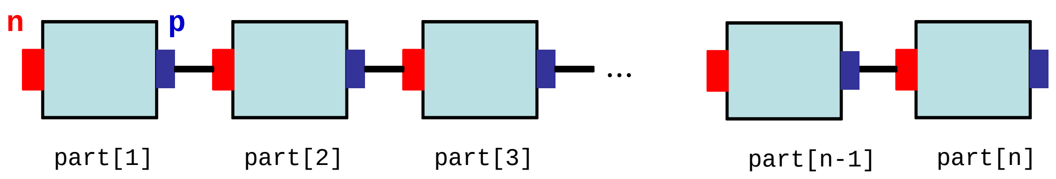
Another feature of Modelica will be described in this section: the **vectors of components**. The concept is illustrated in Figure 3.5. In the upper part of it, the series connection of n components of the same class is depicted. This class, named `Device`, is a model with two connectors, named `p` and `n`. The connection is made so that the `p` connector of the i-th component is connected to the `n` connector of the $(i+1)$-th component.

The fragment of Modelica code written in the lower part of Figure 3.5 describes the model depicted in the upper part of the figure. The code contains the declaration of a component vector, and connect sentences within a for clause. The component vector, named `part`, is composed of `n` components of the `Device` class: `part[1]`, `part[2]`, ..., `part[n]`. The value of the `n` parameter can be changed when the `SeriesParts` class is inherited and instantiated. As the number of model equations depends on `n`, its value cannot be changed after translating the model and, therefore, cannot be changed at the experiment definition.

The use of vectors is illustrated in the following example. Suppose that we want to model the longitudinal heat conduction in a bar. The objective is to analyze the evolution of the temperature distribution along the bar under the following conditions: the temperature at one end of the bar (T_a) changes abruptly, while the temperature at the other end (T_b) is maintained at a constant value of 300 K. We will develop two different models of this system, employing vectors of variables in the first model and vectors of components in the second one.

Let's define N equidistant points along the bar, so that the first and last points are located at the bar ends. If the bar length is L, the distance between two consecutive point is $\Delta x = L/(N-1)$. The temperature at the first point is T_a and the temperature at the N-th point is T_b. Both are known. The temperatures at the inner points are named $T[1], \ldots, T[N-2]$.

Now, let's divide the bar into N elements (control volumes) as follows: the elements located at the bar ends have length $\Delta x/2$, and the inner elements have length Δx. The points with temperatures $T[1], \ldots, T[N-2]$ are located at the center of the inner elements. Let's assume that the first element is at T_a temperature, the second element at $T[1]$, the third element at $T[2]$, ..., the $(N-1)$-th element is at $T[N-2]$ and the N-th element is at T_b temperature. The rate of heat transfer between consecutive elements is represented as q_i. The discretization for $N = 7$ is shown in Figure 3.6.



```
model SeriesParts
   parameter Integer n = 10;
   Device part[n];
equation
   for i in 1:n-1 loop
      connect( part[i].p, part[i+1].n );
   end for;
...
```

Figure 3.5: Model with a regular structure and Modelica code.

The model is described in Modelica Code 3.5. It is composed of the equations that describe the energy conservation of the inner elements, the heat conduction between adjacent elements, and the equations describing the evolution of T_a and T_b. The temperatures of the inner elements and the heat transfer rates are represented by the `T` and `q` vectors of real variables.

The initial temperature of the inner elements is set in an **initial equation** section. The equations written in this section are initial conditions. Therefore, the initial value of `T[1]`, ..., `T[N-2]` is `Tstep1`. Modelica provides complementary ways of setting the initial conditions: using the **start** and **fixed** attributes at the variable declarations; and writing equations within **initial equation** sections and assignments within **initial algorithm** sections. This will be explained in Section 7.6.

An equivalent model is obtained representing the boundary conditions as temperature sources, and the bar as a connection of thermal resistors and capacitors (see an example in Figure 3.7). Thermal resistors represent the heat conduction between adjacent control volumes and thermal capacitors describe the energy balances of control volumes. The model is described in Modelica Code 3.6 and 3.7.

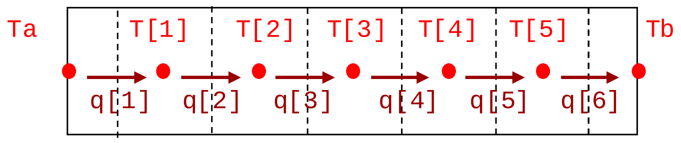


Figure 3.6: Discretization of the spatial coordinate with $N = 7$.

```
model HeatConductionA
   parameter Integer N = 7;
   Real Ta     (unit="K");
   Real Tb     (unit="K");
   Real T[N-2] (unit="K");
   Real q[N-1] (unit="W");
   parameter Real L      (unit="m")          = 1;
   parameter Real S      (unit="m2")         = 0.01;
   parameter Real rho    (unit="kg/m3")      = 7870;
   parameter Real Cp     (unit="J/(kg.K)")   = 449;
   parameter Real k      (unit="W/(m.K)")    = 80;
   parameter Real Tstep1 (unit="K")          = 300;
   parameter Real Tstep2 (unit="K")          = 350;
   parameter Real tstep  (unit="s")          = 50;
   parameter Real Deltax = L / (N-1);
equation
   // Boundary conditions
   Ta = if time < tstep then Tstep1 else Tstep2;
   Tb = Tstep1;
   // Bar
   for i in 1:N-2 loop
      S*Deltax*rho*Cp*der(T[i]) = q[i] - q[i+1];
   end for;
   q[1] = S*k*(Ta-T[1])/Deltax;
   for i in 2:N-2 loop
      q[i] = S*k*(T[i-1]-T[i])/Deltax;
   end for;
   q[N-1] = S*k*(T[N-2]-Tb)/Deltax;
initial equation
   for i in 1:N-2 loop
      T[i] = Tstep1;
   end for;
end HeatConductionA;
```

Modelica Code 3.5: Atomic model of the longitudinal heat conduction in a bar.

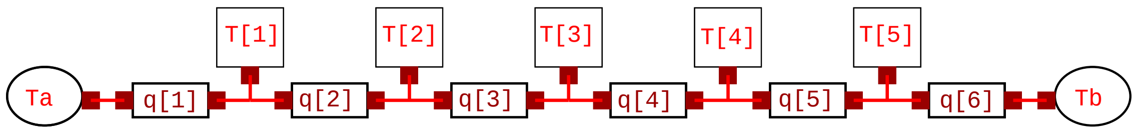


Figure 3.7: Heat conduction in a bar modeled connecting temperature sources, and thermal resistors and capacitors.

```
package ThermalLib

connector ThermalPin
   Real T      (unit="K");
   flow Real q (unit="W");
end ThermalPin;

model ThermalResistor
   ThermalPin p1, p2;
   parameter Real S      (unit="m2");
   parameter Real k      (unit="W/(m.K)");
   parameter Real Deltax (unit="m");
equation
   p1.q = S*k*(p1.T-p2.T)/Deltax;
   p2.q = -p1.q;
end ThermalResistor;

model ThermalCapacitor
   parameter Real S      (unit="m2");
   parameter Real rho    (unit="kg/m3");
   parameter Real Cp     (unit="J/(kg.K)");
   parameter Real Deltax (unit="m");
   parameter Real Tinitial (unit="K");
   ThermalPin p;
equation
   S*Deltax*rho*Cp*der(p.T) = p.q;
initial equation
   p.T = Tinitial;
end ThermalCapacitor;

model SourceT
   parameter Real Tstep1 (unit="K");
   parameter Real Tstep2 (unit="K");
   parameter Real tstep  (unit="s");
   ThermalPin p;
equation
   p.T = if time < tstep
         then Tstep1
         else Tstep2;
end SourceT;

end ThermalLib;
```

Modelica Code 3.6: Thermal library to model the heat conduction in a bar.

```
model HeatConductionB
   parameter Integer N = 7;
   parameter Real L      (unit="m")         = 1;
   parameter Real S      (unit="m2")        = 0.01;
   parameter Real rho    (unit="kg/m3")     = 7870;
   parameter Real Cp     (unit="J/(kg.K)") = 449;
   parameter Real k      (unit="W/(m.K)")  = 80;
   parameter Real Tstep1 (unit="K")         = 300;
   parameter Real Tstep2 (unit="K")         = 350;
   parameter Real tstep  (unit="s")         = 50;
   parameter Real Deltax = L / (N-1);
   ThermalLib.SourceT Ta(Tstep1=Tstep1,Tstep2=Tstep2,tstep=tstep);
   ThermalLib.SourceT Tb(Tstep1=Tstep1,Tstep2=Tstep1,tstep=tstep);
   ThermalLib.ThermalResistor Rth[N-1] (S=fill(S,N-1), k=fill(k,N-1),
                                        Deltax=fill(Deltax,N-1));
   ThermalLib.ThermalCapacitor   Cth[N-2] (S=fill(S,N-2), rho=fill(rho,N-2),
                                        Cp=fill(Cp,N-2), Deltax=fill(Deltax,N-2),
                                        Tinitial=fill(Tstep1,N-2));
equation
   connect (Ta.p, Rth[1].p1);
   for i in 1:N-2 loop
      connect (Rth[i].p2, Cth[i].p);
      connect (Cth[i].p,  Rth[i+1].p1);
   end for;
   connect (Rth[N-1].p2, Tb.p);
end HeatConductionB;
```

Modelica Code 3.7: Composed model of the longitudinal heat conduction in a bar.

3.5 Control of level and temperature in a tank

Consider the system depicted in Figure 3.8. It is composed of a liquid storage tank with a hole in the bottom, a source of liquid and a level control system (labeled as "LC" in the figure). The tank is filled by the source, and drained by gravity through the hole in the bottom. The control system measures the height of liquid in the tank (h) and generates a voltage (u_h) that is applied to the source of liquid. The mass flow of liquid produced by the source (F_{in}) is proportional to the applied input voltage (u_h) if the voltage is positive, and is zero if the voltage is negative.

The liquid level (h) is controlled using a PI controller, this is, a controller with both proportional and integral control. The controller has two inputs: the setpoint (desired or target value for the controlled variable) and the actual value of the controlled variable. The controller calculates the error signal (e), defined as the difference between the setpoint (h_{ref}) and the actual value of the controlled variable (h). The controller output (u_h) is calculated as the sum of two terms: one directly proportional to the error signal, and the other proportional to the integral of the error signal. The PI controller is described by the following equations:

$$e = h_{ref} - h \tag{3.7}$$

$$u_h = \underbrace{k_P \cdot e}_{\substack{\text{Proportional}\\ \text{term}}} + \underbrace{\frac{1}{k_I} \cdot \int_0^t e \cdot dt}_{\substack{\text{Integral}\\ \text{term}}} \tag{3.8}$$

Let I denote the integral of the error signal. Eqs. (3.7) and (3.8) can be written as follows:

$$e = h_{ref} - h \tag{3.9}$$

$$\frac{dI}{dt} = e \tag{3.10}$$

$$u_h = k_P \cdot e + \frac{1}{k_I} \cdot I \tag{3.11}$$

The relevant physical quantities of the system are listed in Table 3.2. The gravitational acceleration (g), the cross-sectional area of the tank (A) and the hole (a), the parameters of the source (k_f) and the PI controller (k_P, k_I), and the liquid density

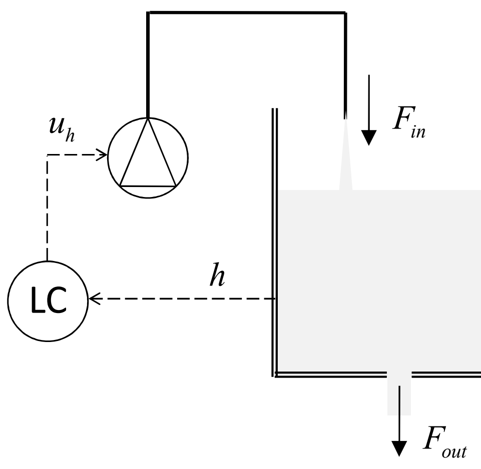


Figure 3.8: Level control in a tank with a hole in the bottom.

Table 3.2: Physical quantities of the system depicted in Figure 3.8.

Symbol	**Quantity**	**Units**	**Value**
a	Cross-sectional area of the hole	m^2	0.1
A	Cross-sectional area of the tank	m^2	2.0
e	Error signal of level controller	m	
F_{in}, F_{out}	Mass flow rates	kg/s	
g	Gravitational acceleration	m/s^2	9.81
h	Height of liquid in the tank	m	
h_{ref}	Liquid level setpoint	m	
I	Integral of the e error signal	m·s	
k_f	Source's coefficient of proportionality	kg/(s·V)	100
k_I	Integral parameter of PI controller	m·s/V	15
k_P	Proportional parameter of PI controller	V/m	2
m	Mass of the liquid inside the tank	kg	
u_h	Controller output	V	
ρ	Liquid density	kg/m^3	760

(ρ) have constant known values. The time-dependent quantities of the system (m, h, F_{out}, F_{in}, e, I, u_h, h_{ref}) can be calculated from Eqs. (3.12) – (3.19).

$$\begin{aligned}
\frac{dm}{dt} &= F_{in} - F_{out} && (3.12)\\
m &= \rho \cdot A \cdot h && (3.13)\\
F_{out} &= a \cdot \rho \cdot \sqrt{2 \cdot g \cdot h} && (3.14)\\
F_{in} &= \max\left(0, k_f \cdot u_h\right) && (3.15)\\
e &= h_{ref} - h && (3.16)\\
\frac{dI}{dt} &= e && (3.17)\\
u_h &= k_P \cdot e + \frac{1}{k_I} \cdot I && (3.18)\\
h_{ref} &= \begin{cases} 5\text{ m} & \text{si } \; t < 300\text{ s} \\ 3\text{ m} & \text{si } \; 300\text{ s} \le t < 600\text{ s} \\ 7\text{ m} & \text{si } \; t \ge 600\text{ s} \end{cases} && (3.19)
\end{aligned}$$

The mass balance in the tank is described by Eq. (3.12). The relationship between the liquid mass and height is described by Eq. (3.13). The mass flow rate of liquid flowing out of the orifice depends on the liquid height as described by Eq. (3.14). The constitutive relationship of the liquid source is Eq. (3.15), and Eqs. (3.16) – (3.18) describe the PI controller. The liquid level setpoint is given by Eq. (3.19).

The initial state of the system is determined by setting the initial value of the mass of liquid inside the tank and the integral of the error signal. The model is described in Modelica Code 3.8. The result obtained simulating the model during 1000 s is shown in Figure 3.9.

Suppose that we connect to the tank a heating system composed of a heater and a temperature controller. The system is depicted in Figure 3.10. The temperature control system (labeled as TC in the figure) measures the liquid temperature (T) and generates a voltage (u_T) that is applied to the heater input. If u_T is positive, then the thermal power (Q) produced by the heater is proportional to u_T. If u_T is negative, Q is zero.

$$Q = \max(0, k_c \cdot u_T) \qquad (3.20)$$

The temperature of the liquid stored inside the tank can be calculated from the energy balance equation for the stored liquid. We assume that the liquid is perfectly

```
model ControlTank1
  import SI = Modelica.SIunits;
  import Modelica.Math.*;
  parameter SI.Area a=0.1 "Cross-sectional area of the hole";
  parameter SI.Area A=2 "Cross-sectional area of the tank";
  SI.Height e "Controller error signal";
  SI.MassFlowRate Fin "Input mass flow rate";
  SI.MassFlowRate Fout "Output mass flow rate";
  constant SI.Acceleration g=9.81 "Gravitational acceleration";
  SI.Height h "Height of liquid in the tank";
  SI.Height href "Liquid level setpoint";
  Real I(unit="m.s", start=0, fixed=true) "Integral of error signal";
  parameter Real kf(unit="kg/(s.V)") = 100 "Source coefficient";
  parameter Real kI(unit="m.s/V") = 15 "Integral parameter of PI controller";
  parameter Real kP(unit="V/m") = 2 "Proportional parameter of PI controller";
  SI.Mass m(start=1e3, fixed=true) "Mass of the liquid inside the tank";
  SI.Voltage uh "Controller output";
  parameter SI.Density rho=760 "Density of the liquid";
equation
  der(m) = Fin - Fout;
  m = rho*A*h;
  Fout = a*rho*sqrt(2*g*h);
  Fin = max(0, kf*uh);
  e = href - h;
  der(I) = e;
  uh = kP*e + I/kI;
  href = if time < 300 then 5 else if time < 600 then 3 else 7;
end ControlTank1;
```

Modelica Code 3.8: Tank with level control shown in Figure 3.8.

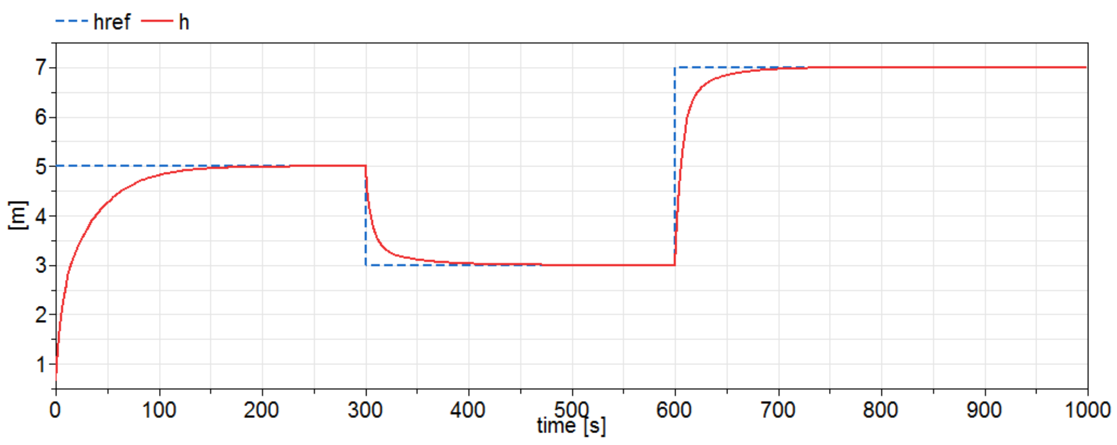


Figure 3.9: Setpoint (h_{ref}) and level (h) obtained simulating Modelica Code 3.8.

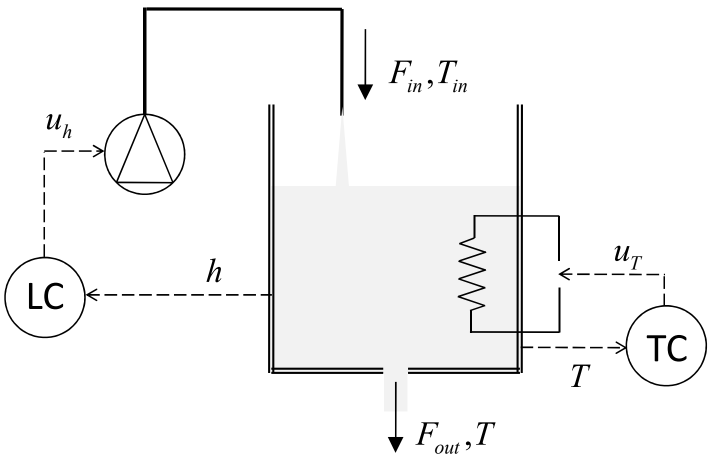


Figure 3.10: Level and temperature control in a tank with a hole in the bottom.

Table 3.3: Physical quantities needed to describe the thermal behavior.

Symbol	**Quantity**	**Units**	**Value**
C_p	Heat capacity of the liquid	J/(kg·K)	
$C_{p,0}$	Zero-order term of C_p	J/(kg·K)	446
$C_{p,1}$	First-order term of C_p	J/(kg·K^2)	5.36
$F_{H,in}$, $F_{H,out}$	Enthalpy flow rates	W	
H	Enthalpy of the liquid stored in the tank	J	
e_T	Error signal of temperature controller	K	
I_T	Integral of error signal e_T	K·s	
k_c	Heater's coefficient	W/V	8E+6
$k_{T,I}$	Integral parameter of temp. controller	K·s/V	50
$k_{T,P}$	Proportional parameter of temp. controller	V/K	0.3
Q	Thermal power produced by the heater	W	
T	Temperature of the liquid inside the tank	K	
T_{in}	Temperature of the input liquid	K	300
T_{ref}	Setpoint of liquid temperature	K	
u_T	Control input to the heater	V	

stirred in the tank, being at uniform temperature T. The temperature of the liquid produced by the source is T_{in}. The liquid exists the tank at temperature T. The energy balance is described in Eq. (3.21).

$$\underbrace{\frac{dH}{dt}}_{\substack{\text{Change in enthalpy}\\\text{of the stored liquid}}} = \underbrace{F_{H,in}}_{\substack{\text{Input enthalpy}\\\text{flow rate}}} - \underbrace{F_{H,out}}_{\substack{\text{Output enthalpy}\\\text{flow rate}}} + \underbrace{Q}_{\substack{\text{Input heat}\\\text{flow rate}}} \tag{3.21}$$

The total enthalpy (H) of the liquid stored in the tank is proportional to the liquid mass (m), temperature (T) and heat capacity at constant pressure (C_p).

$$H = m \cdot C_p \cdot T \tag{3.22}$$

Let's suppose that the heat capacity of the liquid has a linear dependence with the temperature,

$$C_p = C_{p,0} + C_{p,1} \cdot T \tag{3.23}$$

where $C_{p,0}$ and $C_{p,1}$ are known constants. The enthalpy of the stored liquid and the enthalpy flow rates can be calculated as follows:

$$\begin{aligned} H &= m \cdot (C_{p,0} + C_{p,1} \cdot T) \cdot T &(3.24)\\ F_{H,in} &= F_{in} \cdot (C_{p,0} + C_{p,1} \cdot T_{in}) \cdot T_{in} &(3.25)\\ F_{H,out} &= F_{out} \cdot (C_{p,0} + C_{p,1} \cdot T) \cdot T &(3.26)\end{aligned}$$

The PI controller and the temperature setpoint are described as follows:

$$\begin{aligned} e_T &= T_{ref} - T &(3.27)\\ \frac{dI_T}{dt} &= e_T &(3.28)\\ u_T &= k_{T,P} \cdot e_T + \frac{1}{k_{T,I}} \cdot I_T &(3.29)\\ T_{ref} &= \begin{cases} 340\text{ K} & \text{if } t < 500\text{ s} \\ 320\text{ K} & \text{if } t \geq 500\text{ s} \end{cases} &(3.30)\end{aligned}$$

```
model ControlTank2

  import SI = Modelica.SIunits;
  import Modelica.Math.*;

  constant SI.Acceleration g=9.81 "Gravitational acceleration";

  // Liquid
  parameter SI.Density rho=760 "Density";
  parameter SI.SpecificHeatCapacity Cp0=446 "Zero-order term of Cp";
  parameter Real Cp1(unit="J/(kg.K2)") = 5.36 "1st-order coeff. of Cp";

  // Tank
  parameter SI.Area a=0.1 "Cross-sectional area of the hole";
  parameter SI.Area A=2 "Cross-sectional area of the tank";

  // Level controller
  parameter Real kI(unit="m.s/V") = 15 "Integral parameter";
  parameter Real kP(unit="V/m") = 2 "Proportional parameter";

  // Source of liquid
  parameter Real kf(unit="kg/(s.V)") = 100 "Proportionality coefficient";

  // Heater
  parameter Real kc(unit="W/V") = 8E+6 "Proportionality coefficient";

  // Temperature controller
  parameter Real kT_I(unit="K.s/V") = 50 "Integral parameter";
  parameter Real kT_P(unit="V/K") = 0.3 "Proportional parameter";

  SI.Height e "Error signal of level controller";
  SI.MassFlowRate Fin "Input mass flow rate";
  SI.MassFlowRate Fout "Output mass flow rate";
  SI.Height h "Level of liquid";
  SI.Height href "Setpoint of liquid level";
  Real I(unit="m.s", start=0, fixed=true) "Integral of e";
  SI.Mass m(start=1e3, fixed=true) "Mass of stored liquid";
  SI.Voltage uh "Output of level controller";
  SI.EnthalpyFlowRate FHin "Input enthalpy flow rate";
  SI.EnthalpyFlowRate FHout "Output enthalpy flow rate";
  SI.Enthalpy H "Enthalpy of stored liquid";
  SI.Temperature eT "Error signal of temperature controller";
  Real IT(unit="K.s", start=0, fixed=true) "Integral of eT";
  SI.HeatFlowRate Q "Heat flow rate from the heater";
  SI.Temperature T(start=300, fixed=true) "Temperature of stored liquid";
  parameter SI.Temperature Tin=300 "Temperature of input liquid";
  SI.Temperature Tref "Temperature setpoint";
  SI.Voltage uT "Output of temperature controller";
```

Modelica Code 3.9: Atomic model of level and temperature control in a tank (1/2).

```
equation

  // Tank
  der(m) = Fin - Fout;
  der(H) = FHin - FHout + Q;
  m = rho*A*h;
  H = m*(Cp0 + Cp1*T)*T;
  Fout = a*rho*sqrt(2*g*h);
  FHout = Fout*(Cp0 + Cp1*T)*T;

  // Source of liquid
  Fin = max(0, kf*uh);
  FHin = Fin*(Cp0 + Cp1*Tin)*Tin;

  // Level controller
  e = href - h;
  der(I) = e;
  uh = kP*e + I/kI;
  href = if time < 300 then 5 else if time < 600 then 3 else 7;

  // Temperature controller
  eT = Tref - T;
  der(IT) = eT;
  uT = kT_P*eT + IT/kT_I;
  Tref = if time < 500 then 340 else 320;

  // Heater
  Q = max(0,kc*uT);

end ControlTank2;
```

Modelica Code 3.10: Atomic model of level and temperature control in a tank (2/2).

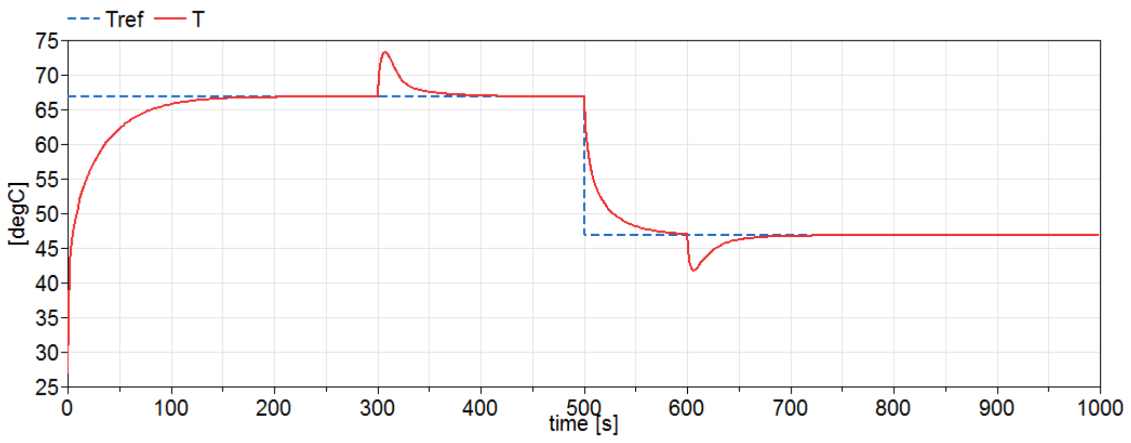


Figure 3.11: Liquid temperature (T) and its setpoint (T_{ref}) obtained simulating Modelica Code 3.9 and 3.10. The liquid level and its setpoint are shown in Figure 3.9.

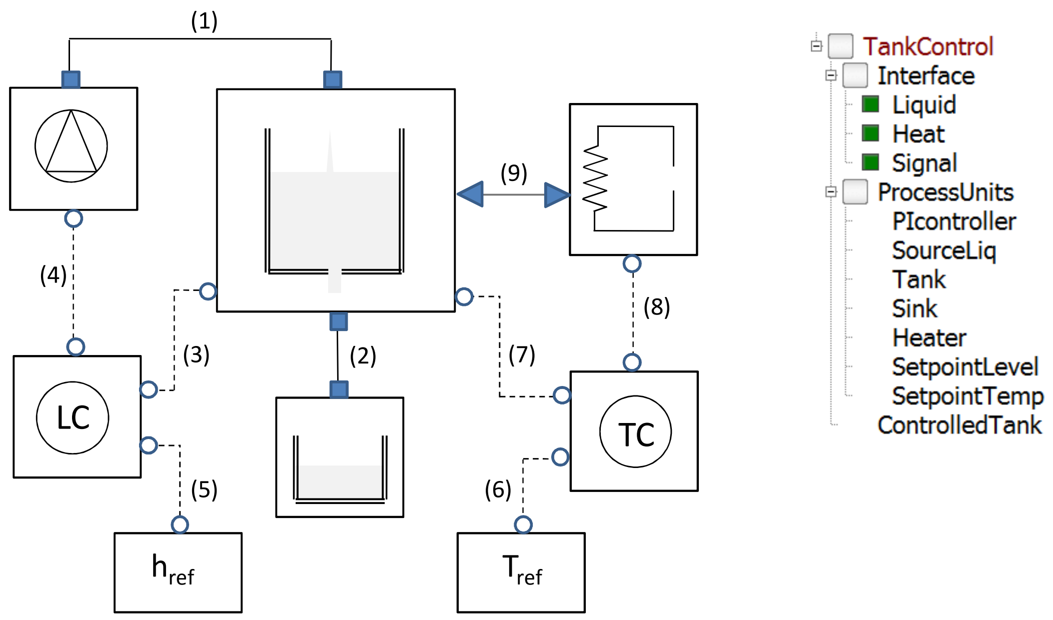


Figure 3.12: System decomposition into parts (left) and library architecture (right).

The initial state of the thermal model is defined by setting the initial temperature of the liquid stored in the tank, and setting I_T to zero. The physical quantities employed to describe the thermal behavior are listed in Table 3.3. The model is described in Modelica Code 3.9 and 3.10. Simulating the model during 1000 s, the temperature of the stored liquid evolves as shown in Figure 3.11.

Let's adopt now another approach. Instead of describing the model equations as an atomic Modelica model, we are going to model the system by applying the object-oriented modeling methodology. The first step is to decompose the system into its parts, deciding which model classes need to be programmed. The system can be decomposed into the following components (see the left side of Figure 3.12): a tank, two PI controllers (LC and TC in the figure), a source of liquid, a heater and two setpoint signal generators (href and Tref in the figure), and a sink.

There are three classes of interaction among the system components: liquid flow, heat transfer and signal transmission. A different symbol has been used in Figure 3.12 to represent the connectors describing each class of interaction. Connectors describing liquid flow are filled squares, connectors describing heat transfer are filled triangles, and connectors describing signal transmission are hollow circles.

We define a library named `TankControl`, whose architecture is shown on the right side of Figure 3.12. Within the `TankControl` package, two packages (`Interface` and `ProcessUnits`) and the model of the complete system (`ControlledTank`) are defined. The library is described in Modelica Code 3.11 – 3.13.

```
encapsulated package TankControl

  import SI = Modelica.SIunits;
  import Modelica.Math.*;

package Interface

  connector Liquid
    flow SI.MassFlowRate Fm     "Mass flow rate";
    flow SI.EnthalpyFlowRate FH "Enthalpy flow rate";
  end Liquid;

  connector Heat
    flow SI.HeatFlowRate Q "Heat flow rate";
  end Heat;

  connector Signal
    Real s;
  end Signal;

end Interface;

package ProcessUnits

  model PIcontroller
      Interface.Signal y;
      Interface.Signal ref;
      Interface.Signal u;
      parameter Real kI "Integral parameter";
      parameter Real kP "Proportional parameter";
    protected
      Real e;
      Real I(start=0, fixed=true) "Integral of e";
  equation
      e = ref.s - y.s;
      der(I) = e;
      u.s = kP*e + I/kI;
  end PIcontroller;

  model SourceLiq
      Interface.Signal u;
      Interface.Liquid portLiq;
      parameter SI.Temperature T "Temperature of liquid";
      parameter Real kf(unit="kg/(s.V)") "Coefficient of proportionality";
      parameter SI.SpecificHeatCapacity Cp0 "Zero-order term of Cp";
      parameter Real Cp1(unit="J/(kg.K2)") "First-order coeff. of Cp";
  equation
      portLiq.Fm = -max(0, kf*u.s);
      portLiq.FH = portLiq.Fm*(Cp0 + Cp1*T)*T;
  end SourceLiq;
```

Modelica Code 3.11: Composed model of level and temperature control in a tank (1/3).

```
model Tank
    Interface.Liquid portLiqSup;
    Interface.Liquid portLiqBase;
    Interface.Heat portCalor;

    Interface.Signal signal_h;
    Interface.Signal signal_T;

    constant SI.Acceleration g=9.81 "Gravitational acceleration";
    parameter SI.Area a "Cross-sectional area of the hole";
    parameter SI.Area A "Cross-sectional area of the tank";
    parameter SI.Density rho "Density of the liquid";
    parameter SI.SpecificHeatCapacity Cp0 "Zero-order term of Cp";
    parameter Real Cp1(unit="J/(kg.K2)") "First-order coeff. of Cp";

    SI.Temperature T(start=300, fixed=true) "Temperature of stored liquid";
    SI.Height h "Level of stored liquid";
    SI.Mass m(start=1e3, fixed=true) "Mass of stored liquid";
    SI.Enthalpy H "Enthalpy of stored liquid";

equation
    der(m) = portLiqSup.Fm + portLiqBase.Fm;
    der(H) = portLiqSup.FH + portLiqBase.FH + portCalor.Q;
    m = rho*A*h;
    H = m*(Cp0 + Cp1*T)*T;
    portLiqBase.Fm = -a*rho*sqrt(2*g*h);
    portLiqBase.FH = portLiqBase.Fm*(Cp0 + Cp1*T)*T;
    signal_h.s = h;
    signal_T.s = T;
end Tank;

model Sink
    Interface.Liquid portLiq;
end Sink;

model Heater
    Interface.Signal cntrl;
    Interface.Heat portCalor;
    parameter Real kc(unit="W/V") "Coefficient of proportionality";
equation
    portCalor.Q = -max(0, kc*cntrl.s);
end Heater;

model SetpointLevel
    Interface.Signal ref;
equation
    ref.s = if time < 300 then 5
               else if time < 600 then 3 else 7;
end SetpointLevel;
```

Modelica Code 3.12: Composed model of level and temperature control in a tank (2/3).

```
  model SetpointTemp
      Interface.Signal ref;
  equation
      ref.s = if time < 500 then 340 else 320;
  end SetpointTemp;

end ProcessUnits;

model ControlledTank
    import TankControl.ProcessUnits.*;
    // Liquid
    parameter SI.Density rho=760 "Density of liquid";
    parameter SI.SpecificHeatCapacity Cp0=446 "Zero-order term of Cp";
    parameter Real Cp1(unit="J/(kg.K2)") = 5.36 "First-order coeff. of Cp";
    // Tank
    parameter SI.Area a=0.1 "Cross-sectional area of the hole";
    parameter SI.Area A=2   "Cross-sectional area of the tank";
    // Source
    parameter SI.Temperature Tin=300 "Temperature of input liquid";
    parameter Real kf(unit="kg/(s.V)") = 100 "Coeff. of source";
    // Heater
    parameter Real kc(unit="W/V") = 8E+6 "Coeff. of heater";
    // Level controller
    parameter Real kI(unit="m.s/V") = 15 "Integral parameter of LC";
    parameter Real kP(unit="V/m") = 2    "Proportional parameter of LC";
    // Temperature controller
    parameter Real kT_I(unit="K.s/V") = 50 "Integral parameter of TC";
    parameter Real kT_P(unit="V/K") = 0.3  "Proportional parameter of TC";

    // Declaration of components
    SourceLiq     source(kf=kf,Cp0=Cp0,Cp1=Cp1,T=Tin);
    Heater        heater(kc=kc);
    Tank          tank(a=a,A=A,rho=rho,Cp0=Cp0,Cp1=Cp1);
    PIcontroller  LC(kI=kI, kP=kP);
    PIcontroller  TC(kI=kT_I, kP=kT_P);
    Sink          sink;
    SetpointLevel levelSP;
    SetpointTemp  tempSP;

equation
    connect (tank.portLiqSup,  source.portLiq); // (1)
    connect (tank.portLiqBase, sink.portLiq);   // (2)
    connect (tank.signal_h,    LC.y);           // (3)
    connect (LC.u,             source.u);       // (4)
    connect (levelSP.ref,      LC.ref);         // (5)
    connect (tempSP.ref,       TC.ref);         // (6)
    connect (tank.signal_T,    TC.y);           // (7)
    connect (heater.cntrl,     TC.u);           // (8)
    connect (heater.portCalor, tank.portCalor); // (9)
end ControlledTank;

end TankControl;
```

Modelica Code 3.13: Composed model of level and temperature control in a tank (3/3).

If you compare Figure 3.10 with Figure 3.12, you may wonder why the `Sink` component is included in the composed model. The reason is that if a connector is left unconnected, the modeling environment automatically adds to the model equations setting to zero the through variables of this connector. In this system in particular, if the liquid flow connector at the tank bottom is left unconnected, the modeling environment includes in the model two equations setting to zero the mass and enthalpy flow rates through the tank hole.

This rule (i.e., flow variables of unconnected connectors are automatically set to zero) is applied by the modeling environment not only when translates a model, but also when checks it. For instance, if you ask Dymola to check the `SourceLiq` model (see Modelica Code 3.11), you get a message indicating that the model contains 3 unknown variables (`portLiq.Fm`, `portLiq.FH` and `u.s`) and 4 equations: the two equations written in the equation section of the class

```
portLiq.Fm = -max(0, kf*u.s);
portLiq.FH = portLiq.Fm*(Cp0 + Cp1*T)*T;
```

and the two equations that Dymola has automatically added:

```
portLiq.Fm = 0;
portLiq.FH = 0;
```

If a model is conceived to be used in a computational causality context different to the context imposed during the check test, obtaining a message indicating that the model is structurally singular should not be understood as indication of an error. In the case of the `SourceLiq` model, it was designed assuming that both connectors will be connected, the two equations of the model will be employed to calculate `portLiq.Fm` and `portLiq.FH`, and `u.s` will be calculated from the connection equation of the `u` connector.

A final comment on a different topic. Modelica models usually contain **annotation** sentences, where the model developer includes the definition of the graphical properties of the icon and the diagram of the model, documentation in HTML format, etc. Although standardization has been achieved in the annotation syntax, the actual support to annotations depends on the modeling environment. The use of annotations will not be discussed in this text. The interested reader may find useful to inspect the documentation of the modeling environment and the source code of the libraries provided by the modeling environment.

3.6 Dissipation of heat generated in a circuit

Up to this point, we have seen two ways of accessing the variables of a component. Interface variables, this is, variables declared within connectors, can be accessed by **connecting connectors**. On the other hand, **dot notation** allows to access interface variables, and local variables not declared as protected. As will be explained in this section, Modelica provides a third way of accessing a component: the inner/outer construct.

Consider the following example. The `Circuit` class is composed of three components of the `Capacitor` class, named `C1`, `C2` and `C3`.

```
model Circuit
   Capacitor C1, C2, C3;
   ...
end Circuit;
```

It is assumed that all circuit components are at the same temperature as the ambient air, T_0. Assuming that the T_0 variable is declared in the `Circuit` and `Capacitor` classes, the condition on temperature can be described using dot notation as follows.

```
model Capacitor
   Real T0;  // Capacitor temperature
   ...
end Capacitor;

model Circuit
   Real T0;  // Environment temperature
   Capacitor C1, C2, C3;
   ...
equation
   C1.T0 = T0;
   C2.T0 = T0;
   C3.T0 = T0;
end Circuit;
```

The use of dot notation implies in this case writing as many equations as components. If an additional capacitor is connected to the circuit, an additional equation equaling the capacitor and environment temperatures has to be written in the circuit model. This does not facilitate the graphical model edition, dragging and dropping components from a model library.

The **inner/outer construct** was introduced in Modelica to facilitate equaling variables of a class with variables of its inner components. Taking advantage of this feature, the circuit model can be described as follows.

```
model Capacitor
   outer Real T0;   // Capacitor temperature
   ...
end Capacitor;

model Circuit
   inner Real T0;   // Environment temperature
   Capacitor C1, C2, C3;
   ...
end Circuit;
```

An inner variable named `T0` is declared in `Circuit`. Outer variables named `T0` are declared in the components of `Circuit`. The modeling environment recognizes that all these variables are alias and automatically writes the equations equaling them.

The same principle can be applied to connectors, so that **inner/outer connectors** are automatically recognized as alias. Consider the following example.

```
connector HeatTransfer
   ...
end HeatTransfer;

model Capacitor
   outer HeatTransfer Qenv;
   HeatTransfer       Qcomp;
   ...
equation
   connect(Qcomp, Qenv);
   ...
end Capacitor;

model Circuit
   inner HeatTransfer Qenv;
   Capacitor C1, C2, C3;
   ...
end Circuit;
```

A connector named `Qenv` has been declared as outer in `Capacitor`, and as inner in `Circuit`. Therefore, the modeling environment recognizes that the `Qenv` connectors of `C1`, `C2` and `C3` are alias of the `Qenv` connector of `Circuit`. As the connect sentence is written in the `Capacitor` class, declaring a component of the `Capacitor` class implies establishing the connection.

Modelica also supports **inner/outer functions**. Let's see an example. Suppose that the temperature T_0 of the ambient air that surrounds the circuit depends on position. It is described by a function that, given the three-dimensional coordinates of a point, returns the temperature at that point. The interface of the function is defined as follows.

```
partial function InterfTempAmbient
   input  Real r[3];
   output Real T;
end InterfTempAmbient;
```

Suppose that N different functions describing the spatial dependence of the ambient temperature are defined. These functions are subclasses of `InterfTempAmbient`.

```
function TempAmbient1               |    ...    | function TempAmbientN
  extends InterfTempAmbient;        |           |   extends InterfTempAmbient;
algorithm                           |           | algorithm
  T := ...                          |           |   T := ...
end TempAmbient1;                   |           | end TempAmbientN;
```

The function that describes the ambient temperature is the same for all circuit components. A convenient way of specifying the function as follows.

```
model Capacitor
   outer function AmbientTemp = InterfTempAmbient;
   Real r[3]; // Capacitor position
   Real T0;   // Temperature at the capacitor position
   ...
equation
   r  = ...
   T0 = AmbientTemp(r);
   ...
end Capacitor;

model Circuit
   inner function AmbientTemp = TempAmbient1;
   Capacitor C1, C2, C3;
   ...
end Circuit;
```

The `AmbientTemp` function is defined as outer in `Capacitor`, and as inner in `Circuit`. Therefore, for the `Capacitor` components defined within `Circuit`, the modeling environment recognizes that these functions are alias.

In the declaration of `AmbientTemp` in `Capacitor`, `InterfTempAmbient` is written on the right hand side of the equal symbol. Writing this, the model developer imposes a condition to be fulfilled by any function assigned to `AmbientTemp`: the function must be a subclass of `InterfTempAmbient`. In this example, `TempAmbient1` is assigned to `AmbientTemp` (see the declaration of `AmbientTemp` in `Circuit`).

3.7 Further reading

The development and use of Modelica libraries are discussed in (ModelicaTM 2000), (Otter 2009), (Fritzson 2011) and (Tiller 2001). The reading of these books is strongly recommended.

Modelica stream connectors have not been explained in this lesson. Readers are referred to (Franke et al. 2009).

Excellent references on modeling of transport phenomena, matter and energy balances, and chemical processes are (Froment & Bischoff 1979), (Bird et al. 1975), (Incropera & DeWitt 1996), (Ramirez 1989) and (Luyben 1990).

The level and temperature in a tank are controlled in Section 3.5 using PI controllers. An excellent reference on this topic is (Åström & Hagglund 1995).

Part II

Simulation of continuous-time models

Lesson 4

Computational causality

4.1 Introduction
4.2 Classification of the model variables
4.3 Structural singularity
4.4 Partition algorithm
4.5 Overdetermined and underdetermined systems
4.6 Example: simulation of an electrical circuit
4.7 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Relate the following two concepts: equation-based modeling language, and assignment of computational causality.

– Analyze manually the computational causality of small-dimension models.

– Analyze the computational structure of overdetermined and underdetermined DAE systems.

– Write the simulation algorithm of small-dimension, non-singular DAE systems.

4.1 Introduction

As was explained in previous lessons, model behavior may be described in Modelica using equations and algorithms.

An **algorithm** is a sorted sequence of assignments. An **assignment** has the form `variable := expression`. The computational causality of an algorithm is specified in the algorithm itself: the variables calculated from the algorithm (computational outputs) are all those written on the left hand side of the assignments, and the rest of variables intervening in the algorithm are computational inputs.

An **equation** consists of two expressions separated by the equal sign. The way of writing an equation does not determine its computational causality. In general, the computational causality of an equation depends on the equation itself, and also on the other equations and algorithms of the model. For this reason, the computational causality of models described using equations is a global property of the model, which has to be analyzed by considering the complete model.

The assignment of computational causality is automatically performed by the Modelica modeling environments, which determine the variable to evaluate from each equation, and also the evaluation order of the model equations and algorithms.

The objective of this lesson is to discuss the concepts behind the assignment of computational causality. The description of the efficient algorithms that are implemented in the Modelica modeling environments is out of the scope of this introductory textbook. These algorithms typically employ bipartite graphs for representing the model computational structure. For the sake of simplicity, we will employ incidence matrices for representing the model computational structure, instead of bipartite graphs.

The procedure for analyzing the computational causality consists of a sequence of steps that are described in Sections 4.2, 4.3 and 4.4. A method for analyzing overdetermined and underdetermined systems is describe in Section 4.5. Finally, an example of computational causality assignment is discussed in Section 4.6.

4.2 Classification of the model variables

The two following actions are performed before starting the analysis of the computational causality.

1. **Replace derivatives by dummy variables**. Along this book, dummy variable names are constructed by concatenating the "der" prefix and the variable names. For instance, $\frac{dx}{dt}$ would be replaced in all the model equations by a dummy variable named $derx$.

2. **Classify model variables into known and unknown variables**. Unknown variables are those that have to be calculated from the model equations. Therefore, the objective of assigning the computational causality is to find out which equation has to be employed to evaluate each unknown variable.

   The following variables are classified as **known variables**:

   – The **time** variable.

   – The **constants** and **parameters**. Their values are computed at the beginning of the simulation and don't change during the simulation.

   – The **state variables**. They are not calculated from the model equations. The state variables are calculated by numerical integration of their derivatives.

   The following variables are classified as **unknown variables**:

   – The **dummy variables** introduced for replacing the derivatives.

   – The **remaining variables**. This is, the time-dependent model variables that are not selected as state variables.

For the sake of simplicity, let's suppose by now (we will eliminate this assumption in the next lesson) that all variables that appear differentiated can be selected as state variables. This is equivalent to assume that the model has an equal number of degrees of freedom, and differentiated variables.

Attending to this classification of the model variables, we can represent the set of model equations as

$$\mathbf{F}\left(\mathbf{x},\mathbf{y}\right)=0 \tag{4.1}$$

where the $\mathbf{x}$ and $\mathbf{y}$ vectors represent the known and unknown variables respectively.

On the basis of this representation of the model, the **incidence matrix** of the model, also known as **structural Jacobian matrix**, is a matrix of Boolean elements that indicates which unknown variables intervene in each equation. The incidence matrix can be defined as follows.

If the j-th unknown variable does not appear in the i-th equation of the model, then the (i, j) element of the incidence matrix is zero. Otherwise, the element is one, and it is represented by a cross (X).

The following example allows to illustrate the construction of the incidence matrix. Suppose a model composed of three equations, and three variables named x, y and z. The computational structure of the model is as follows:

– x, y and z intervene in the first equation.

– x and y, and the derivative of x, intervene in the second equation.

– x and z intervene in the third equation.

The computational structure of this model is represented as shown below, where $\dot{x}$ represents the derivative of x.

$$\begin{aligned} f_1(x, y, z) &= 0 \quad (4.2) \\ f_2(x, \dot{x}, y) &= 0 \quad (4.3) \\ f_3(x, z) &= 0 \quad (4.4) \end{aligned}$$

Replacing $\dot{x}$ by $derx$, it is obtained:

$$\begin{aligned} f_1(x, y, z) &= 0 \quad (4.5) \\ f_2(x, derx, y) &= 0 \quad (4.6) \\ f_3(x, z) &= 0 \quad (4.7) \end{aligned}$$

As the derivative of x appears in the model, let's assume that x can be selected as state variable. Selecting x as state variable, the model has the following three unknown variables: $derx$, y and z. The incidence matrix is:

$$\begin{array}{c|ccc|} & derx & y & z \\ f_1 & 0 & X & X \\ f_2 & X & X & 0 \\ f_3 & 0 & 0 & X \end{array} \quad (4.8)$$

The incidence matrix obtained directly from the model, before any manipulation is applied, is named the **original incidence matrix** of the model.

4.3 Structural singularity

The next step in the analysis of the model computational causality is to check whether the model is structurally singular. To this end, it is checked whether:

1. The number of unknown variables is the same as the number of equations.
2. Each unknown variable can be associated with an equation that satisfies the following two conditions simultaneously: the variable intervenes in the equation, and another unknown variable has not been associated to this equation previously.

This latter condition is checked by finding a sequence of permutations of the matrix columns (unknown variables) that allows to obtain a permuted matrix with all the elements on the main diagonal different from zero. If such a sequence of permutations does not exist, then the model is said to be **structurally singular**.

For instance, permuting the order of the two first columns of the incidence matrix shown in (4.8), it is obtained a matrix with all diagonal elements different from zero. Therefore, the model represented by the matrix is not structurally singular.

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccc} derx & y & z \end{array} \\ \left(\begin{array}{ccc} 0 & X & X \\ X & X & 0 \\ 0 & 0 & X \end{array} \right) \end{array} \quad \rightarrow \quad \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccc} y & derx & z \end{array} \\ \left(\begin{array}{ccc} X & 0 & X \\ X & X & 0 \\ 0 & 0 & X \end{array} \right) \end{array} \tag{4.9}$$

The following incidence matrix corresponds to a structurally singular model.

$$\begin{array}{l} f_1(x,z) = 0 \\ f_2(x,\dot{x},y) = 0 \\ f_3(x) = 0 \end{array} \quad \rightarrow \quad \begin{array}{l} f_1(x,z) = 0 \\ f_2(x,derx,y) = 0 \\ f_3(x) = 0 \end{array} \quad \rightarrow \quad \begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccc} derx & y & z \end{array} \\ \left(\begin{array}{ccc} 0 & 0 & X \\ X & X & 0 \\ 0 & 0 & 0 \end{array} \right) \end{array} \tag{4.10}$$

This model contains only one equation (the equation named f_2) to calculate two variables ($derx$ and y), and one equation (the equation named f_3) does not contain any unknown variable. As x appears differentiated in the model, we have assumed that x is a state variable and, therefore, it is classified as a known variable: it is not calculated from the model equations, but by numerical integration of $derx$.

In general, models are structurally singular due to the following reasons:

1. The *number of equations (E) is equal to the number of unknown variables (V)*, but it is not possible to obtain a matrix with zero-free main diagonal by permuting the columns of the incidence matrix. An example of structurally singular model with $E = V$ is shown in (4.10). Given this situation, there exist two possibilities:

   a) The model is *mathematically incorrect.*

   b) The model is mathematically correct, but *the number of its degrees of freedom is smaller than the number of variables that appear differentiated in the model.* This will be explained in Lesson 5.

2. The *number of equations (E) and the number of unknown variables (V) are not equal.* In this case, the model is said to be **overdetermined** ($E > V$) or **underdetermined** ($E < V$). The analysis of these types of model will be discussed in Section 4.5.

4.4 Partition algorithm

Let's suppose that the model is not structurally singular. This is equivalent to suppose that, by permuting the columns of the incidence matrix, it is possible to obtain a matrix with no zeros in the main diagonal. The next step is to transform the incidence matrix into a block lower triangular (BLT) matrix, with diagonal blocks as smaller as possible. This transformation, known as **partitioning the model**, is made by permuting rows and permuting columns of the incidence matrix. The incidence matrix written in BLT form is named the **sorted incidence matrix** of the model.

The incidence matrix in BLT form has the following property. The unknown variables that intervene in the equations of each diagonal block are calculated from the equations in this block, or from the equations in previous blocks.

In the particular case in which the sorted incidence matrix has all its diagonal blocks with dimension 1×1, the model variables can be calculated in sequence, one after another, using one equation to calculate each unknown variable. A diagonal block with one element, (E_i, V_i), indicates that the V_i variable has to be calculated from the E_i equation. The other variables that intervene in the E_i equation (in general, $V_1, \dots, V_{i-1}$) are calculated from the $E_1, \dots, E_{i-1}$ equations. If the variable

V_i intervenes linearly in E_i, this equation can be manipulated symbolically to obtain an explicit expression for V_i. If V_i intervenes non-linearly in E_i, a root-finding algorithm (e.g., the Newton's method) needs to be used.

Diagonal blocks of size $N \times N$ indicate that the corresponding set of N variables (those represented by the block's columns) is calculated solving a system of N simultaneous equations (those represented by the block's rows). The symbolic or numerical method employed for solving the system of simultaneous equations does not concern to the partition algorithm.

Several algorithms have been proposed to transform the incidence matrix into its BLT form. For example, a widely-used algorithm is the **Tarjan's algorithm**, which employs bipartite graphs to represent the computational structure of the model. The description of the Tarjan's algorithm is out of the scope of this text.

For the sake of simplicity, we will describe a procedure well-suited for partitioning manually models with a small number of equations. The procedure consists in repeatedly applying the following rules:

Rule 1. Suppose that one or more unknown variables appear in an equation, and only one of these variables has not been evaluated yet. Then, the equation must be employed to calculate this unevaluated unknown variable.

Rule 2. Suppose that an unknown variable has not been evaluated yet, and this variable only appears in one of the equations whose computational causality has not been assigned yet. Then, the unevaluated unknown variable must be calculated from this equation.

The following example allows to illustrate this procedure for partitioning the model. Let's consider again the model with incidence matrix (4.8). The model and the original incidence matrix are shown again for the reader's convenience.

$$\begin{array}{l} f_1(x,y,z)=0 \\ f_2(x,\dot{x},y)=0 \\ f_3(x,z)=0 \end{array} \quad \rightarrow \quad \begin{array}{l} f_1(x,y,z)=0 \\ f_2(x,derx,y)=0 \\ f_3(x,z)=0 \end{array} \quad \rightarrow \quad \begin{array}{c|ccc} & derx & y & z \\ \hline f_1 & 0 & X & X \\ f_2 & X & X & 0 \\ f_3 & 0 & 0 & X \end{array} \tag{4.11}$$

1. The f_3 equation contains only one unknown variable: the z variable. Therefore, f_3 must to be employed to evaluate z (Rule 1). As z not only appears in f_3, but

it also appears in f_1, the value of z is required for calculating other unevaluated variables. So, f_3 is moved to the first row and z to the first column.

The $derx$ variable appears in only one equation: the f_2 equation. Therefore, $derx$ must be evaluated from f_2 (Rule 2). As $derx$ does not intervene in any other equation, the value of $derx$ is not employed for calculating other unknown variables. So, f_2 is moved to the last row and $derx$ to the last column.

$$\begin{array}{c} \\ f_3 \\ f_1 \\ f_2 \end{array} \begin{array}{c} \begin{array}{ccc} z & y & derx \end{array} \\ \left(\begin{array}{ccc} \boxed{X} & 0 & 0 \\ X & X & 0 \\ 0 & X & \boxed{X} \end{array} \right) \end{array} \tag{4.12}$$

2. As z is evaluated from f_3, f_1 contains only one unevaluated variable: y. Therefore, y must be evaluated from f_1 (Rule 1).

$$\begin{array}{c} \\ f_3 \\ f_1 \\ f_2 \end{array} \begin{array}{c} \begin{array}{ccc} z & y & derx \end{array} \\ \left(\begin{array}{ccc} \boxed{X} & 0 & 0 \\ X & \boxed{X} & 0 \\ 0 & X & \boxed{X} \end{array} \right) \end{array} \tag{4.13}$$

The sorted model, with the computational causality annotated, is shown below. Observe that the variable to be evaluated from each equation is signaled by including the variable within square brackets.

$$f_3(x, [z]) = 0 \tag{4.14}$$
$$f_1(x, [y], z) = 0 \tag{4.15}$$
$$f_2(x, [derx], y) = 0 \tag{4.16}$$

4.5 Overdetermined and underdetermined systems

If the model is **underdetermined**, it is useful to find out which variables can be solved and which ones cannot. To this end, a set of dummy equations is included in the model, satisfying that: the resultant number of equations is equal to the total number of unknown variables; and each dummy equation contains all the unknown variables. This is equivalent to add as many rows filled with ones as required to make the incidence matrix square.

Then, this extended matrix is partitioned. As all the unknown variables intervene in each dummy equation, all these dummy equations will be in the last diagonal block of the sorted incidence matrix. All diagonal blocks except the last one define the unknown variables that can be calculated from the model equations, and the sequence in which these calculations must be performed.

An example is shown below. Let's consider a model composed of two equations, and three variables: x, y, z. The computational structure is as follows. The x, y and z variables, and the derivative of x, appear in the first equation. The x and y variables appear in the second equation. Therefore, the model has three unknown variables ($derx$, y, z) and two equations. In order to analyze which unknown variables can be calculated from the model, a dummy equation is added to the model.

$$\begin{array}{l} f_1(x,\dot{x},y,z)=0 \\ f_2(x,y)=0 \end{array} \quad \rightarrow \quad \begin{array}{cc} & \begin{array}{ccc} derx & y & z \end{array} \\ \begin{array}{r} f_1 \\ f_2 \\ \text{dummy eq.} \end{array} & \begin{pmatrix} \text{X} & \text{X} & \text{X} \\ 0 & \text{X} & 0 \\ \text{X} & \text{X} & \text{X} \end{pmatrix} \end{array} \tag{4.17}$$

Next, the extended incidence matrix is partitioned. As f_2 only contains one unknown variable (y), f_2 is moved to the first row and y to the first column. The incidence matrix in its BLT form is (4.18). As the $derx$ and z variables are calculated in the last diagonal block, one additional equation is needed to calculate these two variables.

$$\begin{array}{cc} & \begin{array}{ccc} y & derx & z \end{array} \\ \begin{array}{r} f_2 \\ f_1 \\ \text{dummy eq.} \end{array} & \left(\begin{array}{c|cc} \boxed{\text{X}} & 0 & 0 \\ \hline \text{X} & \text{X} & \text{X} \\ \text{X} & \text{X} & \text{X} \end{array}\right) \end{array} \tag{4.18}$$

Overdetermined models can be analyzed analogously. Let's denote the difference between the number of equations and unknown variables as $E-V$. In order to analyze the overdetermined model, $E-V$ dummy variables are introduced, so that all intervene in every model equation. This is equivalent to add as many columns filled with ones as required to make the incidence matrix square.

Partitioning this extended incidence matrix, the obtained BLT matrix will have some unknown variables, and all the dummy variables, in the first diagonal block. The reason why the dummy variables appear in the first diagonal block is the following. As all the dummy variables have been included in all the model equations, it is necessary to know the value of the dummy variables for calculating all the unknown variables of the model.

The $E - V$ redundant equations (as many as dummy variables) are among the equations of the first diagonal block. The variables of the first diagonal block are all the dummy variables, and the unknown variables that appear in the redundant equations.

An example is shown below. The model is composed of four equations, and has three variables: x, y, z. The computational structure of the model equations, and the extended incidence matrix, are shown in (4.19). Observe that a dummy variable named α has been included in all equations.

$$\begin{array}{l} f_1(y) = 0 \\ f_2(y,z) = 0 \\ f_3(x,y,z) = 0 \\ f_4(\dot{x},x,z) = 0 \end{array} \quad \rightarrow \quad \begin{array}{c|cccc} & derx & y & z & \alpha \\ \hline f_1 & 0 & X & 0 & X \\ f_2 & 0 & X & X & X \\ f_3 & 0 & X & X & X \\ f_4 & X & 0 & X & X \end{array} \tag{4.19}$$

As $derx$ only appears in f_4, f_4 must be employed to calculate $derx$. The f_4 equation stays in the last row and $derx$ is moved to the last column. The other three unknown variables are evaluated in the first diagonal block. In conclusion, one equation among f_1, f_2 and f_3 is redundant.

$$\begin{array}{c|cccc} & y & z & \alpha & derx \\ \hline f_1 & X & 0 & X & 0 \\ f_2 & X & X & X & 0 \\ f_3 & X & X & X & 0 \\ f_4 & 0 & X & X & \boxed{X} \end{array} \tag{4.20}$$

4.6 Example: simulation of an electrical circuit

The example discussed in this section allows to illustrate the assignment of computational causality. The diagram of an electrical circuit is shown in Figure 4.1. It is composed of a sinusoidal voltage source, two resistors and two capacitors. Names have been assigned to the currents, and the node voltages. It is assumed that the resistances (R_1, R_2), capacitances (C_1, C_2), and the amplitude (U) and frequency

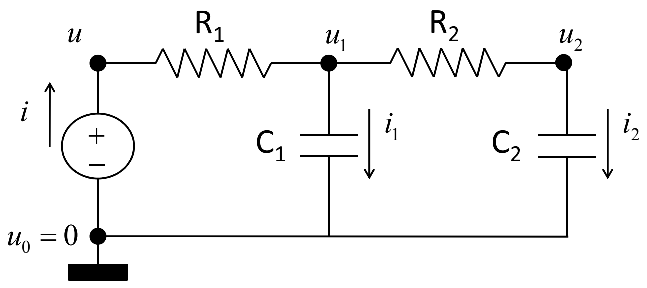


Figure 4.1: Diagram of the electrical circuit.

(w) of the source, are known parameters. The circuit model is composed of the following six equations:

$$u = U \cdot \sin(w \cdot t) \tag{4.21}$$
$$u - u_1 = i \cdot R_1 \tag{4.22}$$
$$C_1 \cdot \frac{du_1}{dt} = i_1 \tag{4.23}$$
$$u_1 - u_2 = i_2 \cdot R_2 \tag{4.24}$$
$$C_2 \cdot \frac{du_2}{dt} = i_2 \tag{4.25}$$
$$i = i_1 + i_2 \tag{4.26}$$

Eqs. (4.21) – (4.25) are the constitutive relationships of the five components, and Eq. (4.26) imposes the current conservation at node u_1.

The first step is to replace in the model the derivatives by dummy variables: $\frac{du_1}{dt} \rightarrow deru_1$, $\frac{du_2}{dt} \rightarrow deru_2$. Making these substitutions, it is obtained:

$$u = U \cdot sin(w \cdot t) \tag{4.27}$$
$$u - u_1 = i \cdot R_1 \tag{4.28}$$
$$C_1 \cdot deru_1 = i_1 \tag{4.29}$$
$$u_1 - u_2 = i_2 \cdot R_2 \tag{4.30}$$
$$C_2 \cdot deru_2 = i_2 \tag{4.31}$$
$$i = i_1 + i_2 \tag{4.32}$$

The next step is to classify the model variables into known variables (time, parameters and state variables) and unknown variables (remaining model variables and dummy variables introduced by replacing the derivatives). Assuming that the two variables that appear differentiated can be selected as state variables, the variables are classified as follows.

– Known: t
U, w, R_1, C_1, R_2, C_2
u_1, u_2

– Unknown: i, i_1, i_2, u
$deru_1$, $deru_2$

The original incidence matrix of the model is built from Eqs. (4.27) – (4.32), taking into account the previous classification of the variables.

$$\begin{array}{r|cccccc} & deru_1 & deru_2 & i & i_1 & i_2 & u \\ u=U\cdot sin(w\cdot t) & 0 & 0 & 0 & 0 & 0 & X \\ u-u_1=i\cdot R_1 & 0 & 0 & X & 0 & 0 & X \\ C_1\cdot deru_1=i_1 & X & 0 & 0 & X & 0 & 0 \\ u_1-u_2=i_2\cdot R_2 & 0 & 0 & 0 & 0 & X & 0 \\ C_2\cdot deru_2=i_2 & 0 & X & 0 & 0 & X & 0 \\ i=i_1+i_2 & 0 & 0 & X & X & X & 0 \end{array} \tag{4.33}$$

With the purpose of analyzing whether the model is structurally singular, it is checked that:

1. The number of equations and the number of unknown variables are equal. This model has six equations, and six unknown variables: $deru_1$, $deru_2$, i, i_1, i_2, u.

2. A one-to-one relationship can be established between the unknown variables and the model equations, satisfying that: each unknown variable is associated with an equation in which the variable appears, and each equation is associated with only one variable. This is equivalent to find a sequence of column permutations that transforms the original incidence matrix into a matrix with no zeros in the diagonal. The obtained matrix is shown below.

$$\begin{array}{r|cccccc|}
 & u & i & deru_1 & i_2 & deru_2 & i_1 \\
u{=}U{\cdot}sin(w{\cdot}t) & X & 0 & 0 & 0 & 0 & 0 \\
u{-}u_1{=}i{\cdot}R_1 & X & X & 0 & 0 & 0 & 0 \\
C_1{\cdot}deru_1{=}i_1 & 0 & 0 & X & 0 & 0 & X \\
u_1{-}u_2{=}i_2{\cdot}R_2 & 0 & 0 & 0 & X & 0 & 0 \\
C_2{\cdot}deru_2{=}i_2 & 0 & 0 & 0 & X & X & 0 \\
i{=}i_1{+}i_2 & 0 & X & 0 & X & 0 & X
\end{array} \tag{4.34}$$

The sorted incidence matrix can be obtained by applying the two rules, as is described next.

1. The constitutive relationship of the voltage source contains only one unknown variable: u. The constitutive relationship of the R_2 resistor contains only one unknown variable: i_2. Therefore, these variables must be calculated from these equations. As these unknown variables intervene in other equations, these two equations are moved to the firsts rows, and these variables to the firsts columns.

   The $deru1$ variable only appears in one equation: the constitutive relationship of the C_1 capacitor. The $deru2$ variable only appears in one equation: the constitutive relationship of the C_2 capacitor. As the values of these variables are not employed in calculating other variables, these two equations are moved to the lasts rows and these variables to the lasts columns.

$$\begin{array}{r|cccccc|}
 & u & i_2 & i & i_1 & deru_1 & deru_2 \\
u{=}U{\cdot}sin(w{\cdot}t) & \boxed{X} & 0 & 0 & 0 & 0 & 0 \\
u_1{-}u_2{=}i_2{\cdot}R_2 & 0 & \boxed{X} & 0 & 0 & 0 & 0 \\
i{=}i_1{+}i_2 & 0 & X & X & X & 0 & 0 \\
u{-}u_1{=}i{\cdot}R_1 & X & 0 & X & 0 & 0 & 0 \\
C_1{\cdot}deru_1{=}i_1 & 0 & 0 & 0 & X & \boxed{X} & 0 \\
C_2{\cdot}deru_2{=}i_2 & 0 & X & 0 & 0 & 0 & \boxed{X}
\end{array} \tag{4.35}$$

2. Assuming that u and i_2 have already been evaluated, the constitutive relationship of the R_1 resistor has only one unevaluated variable: i. Moving this equation to the third row and i to the third column, the following incidence matrix is obtained.

$$
\begin{array}{c|cccccc}
 & u & i_2 & i & i_1 & deru_1 & deru_2 \\
u{=}U{\cdot}sin(w{\cdot}t) & \boxed{X} & 0 & 0 & 0 & 0 & 0 \\
u_1{-}u_2{=}i_2{\cdot}R_2 & 0 & \boxed{X} & 0 & 0 & 0 & 0 \\
u{-}u_1{=}i{\cdot}R_1 & X & 0 & \boxed{X} & 0 & 0 & 0 \\
i{=}i_1{+}i_2 & 0 & X & X & X & 0 & 0 \\
C_1{\cdot}deru_1{=}i_1 & 0 & 0 & 0 & X & \boxed{X} & 0 \\
C_2{\cdot}deru_2{=}i_2 & 0 & X & 0 & 0 & 0 & \boxed{X}
\end{array}
\tag{4.36}
$$

3. Assuming that u, i_2 and i have already been evaluated, the current conservation equation contains only one unevaluated variable: i_1. Therefore, i_1 must be calculated from the current conservation equation. The incidence matrix is now written in BLT form.

$$
\begin{array}{c|cccccc}
 & u & i_2 & i & i_1 & deru_1 & deru_2 \\
u{=}U{\cdot}sin(w{\cdot}t) & \boxed{X} & 0 & 0 & 0 & 0 & 0 \\
u_1{-}u_2{=}i_2{\cdot}R_2 & 0 & \boxed{X} & 0 & 0 & 0 & 0 \\
u{-}u_1{=}i{\cdot}R_1 & X & 0 & \boxed{X} & 0 & 0 & 0 \\
i{=}i_1{+}i_2 & 0 & X & X & \boxed{X} & 0 & 0 \\
C_1{\cdot}deru_1{=}i_1 & 0 & 0 & 0 & X & \boxed{X} & 0 \\
C_2{\cdot}deru_2{=}i_2 & 0 & X & 0 & 0 & 0 & \boxed{X}
\end{array}
\tag{4.37}
$$

Therefore, the sorted model equations, with the computational causality annotated, are the following:

$$
\begin{aligned}
[u] &= U \cdot sin(w \cdot t) && (4.38)\\
u_1 - u_2 &= [i_2] \cdot R_2 && (4.39)\\
u - u_1 &= [i] \cdot R_1 && (4.40)\\
i &= [i_1] + i_2 && (4.41)\\
C_1 \cdot [deru_1] &= i_1 && (4.42)\\
C_2 \cdot [deru_2] &= i_2 && (4.43)
\end{aligned}
$$

Manipulating the equations, the variable to calculate from each equation is isolated on one side of the equation. The sorted and solved model is obtained:

$$
[u] = U \cdot sin(w \cdot t) \tag{4.44}
$$

$$[i_2] = \frac{u_1 - u_2}{R_2} \tag{4.45}$$

$$[i] = \frac{u - u_1}{R_1} \tag{4.46}$$

$$[i_1] = i - i_2 \tag{4.47}$$

$$[deru_1] = \frac{i_1}{C_1} \tag{4.48}$$

$$[deru_2] = \frac{i_2}{C_2} \tag{4.49}$$

Observe that the decision on which variable to evaluate from each equation is unique in this model. However, the equations can be sorted in several equivalent ways. For instance, the order of Eqs. (4.44) and (4.45) can be exchanged. The same applies to Eqs. (4.48) and (4.49).

An algorithm to simulate the circuit model is shown in Figure 4.2. The numerical integration is performed applying the **forward Euler method**. Observe that the classification of the model variables into parameters (time-independent variables), state variables, and algebraic variables (time-dependent variables not selected as state variables) is used as the basis for writing the simulation algorithm.

- The **parameter** values are set at the starting of the algorithm and are kept constant during all the simulation.
- **Initial values** are given to the state variables.
- The **state variables** are calculated by numerical integration of their derivatives. Given the following ordinary differential equation,

  $$\frac{dx}{dt} = f(x,t) \tag{4.50}$$

  the step formula of the forward Euler method is:

  $$x_{i+1} = x_i + f(x_i, t_i) \cdot \Delta t \tag{4.51}$$

  where x_i and x_{i+1} represent the value of the x variable at time t_i and $t_i + \Delta t$ respectively, and $f(x_i, t_i)$ represents the derivative value (i.e., $\frac{dx}{dt}$) at time t_i.
- The values of the **algebraic variables** are calculated from the sorted and solved model, which was obtained by applying the partition algorithm.

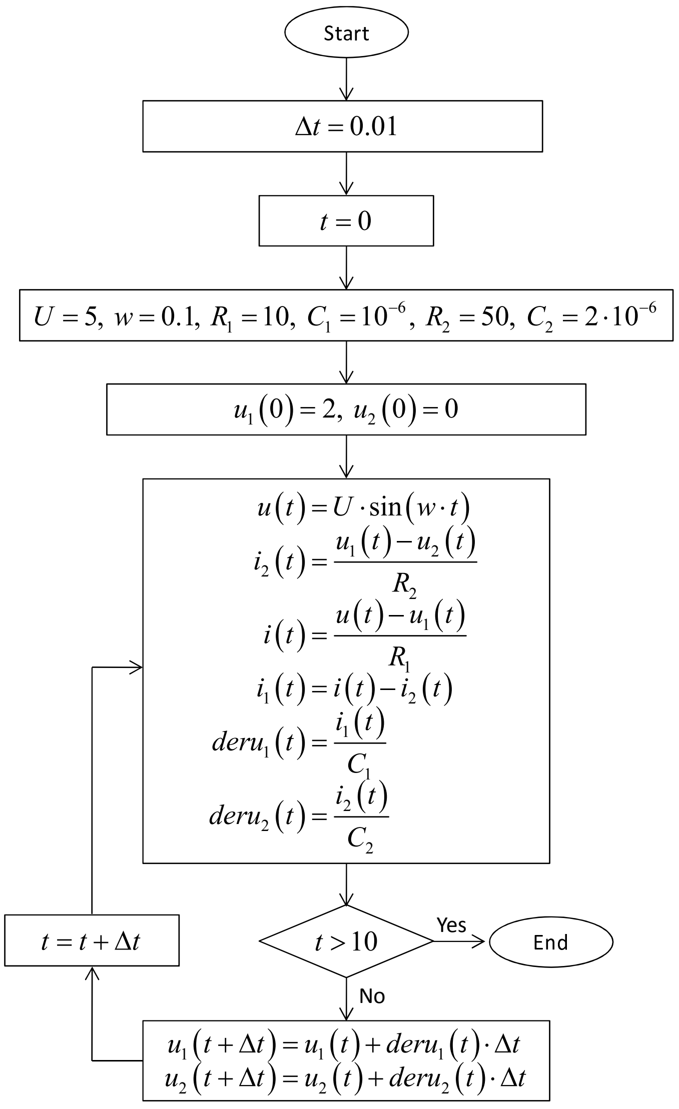

Start
$\Delta t = 0.01$
$t = 0$
$U = 5,\ w = 0.1,\ R_1 = 10,\ C_1 = 10^{-6},\ R_2 = 50,\ C_2 = 2\cdot 10^{-6}$
$u_1(0) = 2,\ u_2(0) = 0$
$u(t) = U\cdot\sin(w\cdot t)$
$i_2(t) = \frac{u_1(t) - u_2(t)}{R_2}$
$i(t) = \frac{u(t) - u_1(t)}{R_1}$
$i_1(t) = i(t) - i_2(t)$
$deru_1(t) = \frac{i_1(t)}{C_1}$
$deru_2(t) = \frac{i_2(t)}{C_2}$
$t > 10$
Yes
End
No
$u_1(t+\Delta t) = u_1(t) + deru_1(t)\cdot\Delta t$
$u_2(t+\Delta t) = u_2(t) + deru_2(t)\cdot\Delta t$
$t = t + \Delta t$

Figure 4.2: Simulation algorithm for the circuit shown in Figure 4.1.

4.7 Further reading

The analysis of the model computational causality is discussed in (Elmqvist 1978). We have used this PhD thesis as main reference in preparing this lesson. The reading of this thesis is strongly recommended. The system structure analysis in other contexts is explained in (Steward 1981).

The analysis of computational causality using bipartite graphs is explained in Chapter 7 of (Cellier & Kofman 2006).

We have not discussed the concept of balanced model in this lesson. The reader is referred to (Olsson et al. 2008).

Lesson 5

Index and initialization of DAE systems

5.1 Introduction
5.2 Structurally singular DAE systems
5.3 Index of DAE systems
5.4 Initialization of DAE systems
5.5 Selection of the state variables
5.6 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Discuss how high-index DAE systems are manipulated to reduce their index by Modelica tools such as Dymola and OpenModelica.

– Calculate the index of small-dimension DAE systems.

– Discuss the difficulties associated to the numerical solution of high-index DAE systems.

– Formulate the initialization problem of small-dimension DAE systems, making explicit (if any) the hidden constraints.

– Manipulate small-dimension DAE systems for selecting the state variables.

– Select the state variables in Modelica.

– Discuss the principles of the dynamic state selection supported by Dymola.

5.1 Introduction

A step in the analysis of the model computational causality is to check whether the model is structurally singular. As explained in Section 4.3, some structurally singular models are mathematically incorrect. However, this is not always the case.

The classification of the model variables into known and unknown variables (see Section 4.2) is made assuming that all the variables that appear differentiated can be selected as state variables. This assumption is incorrect for models with a number of degrees of freedom smaller than the number of variables that appear differentiated. When analyzing the computational causality of these models, it is erroneous to classify as known variables all the variables that appear differentiated.

In this context, the **number of degrees of freedom** (DoF) of a model is the number of time-dependent variables of the model whose initial value can be set independently. For instance, the model depicted in Figure 4.1, whose simulation algorithm is shown in Figure 4.2, has two DoF: the initial value of u_1 and the initial value of u_2 can be set independently. Observe that, known the initial value of these two variables and the value of the parameters, it is possible to calculate, from the model equations, the initial value of the other time-dependent variables (u, i, i_1, i_2, $deru_1$, $deru_2$).

An example is used in Section 5.2 to explain different approaches for solving structurally singular DAE systems, including the approach implemented in Dymola and OpenModelica. The definition of index, and its relationship with the numerical properties of the DAE system, are discussed in Section 5.3. The initialization of DAE systems is discussed in Section 5.4 and the selection of the state variables in Section 5.5.

5.2 Structurally singular DAE systems

The system shown in Figure 5.1 is composed of two liquid storage tanks, connected in parallel to a source of liquid. The symbols p and F_V represent the bottom pressure and the volumetric flow rate respectively, and $f(t)$ is a known function of time. The liquid density (ρ) and the cross-sectional area of the tanks (S_1, S_2) have known constant values. The model parameters C_1 and C_2 are calculated as shown below, where g represents the gravitational acceleration.

$$C_1 = \frac{S_1}{\rho \cdot g} \qquad\qquad C_2 = \frac{S_2}{\rho \cdot g} \tag{5.1}$$

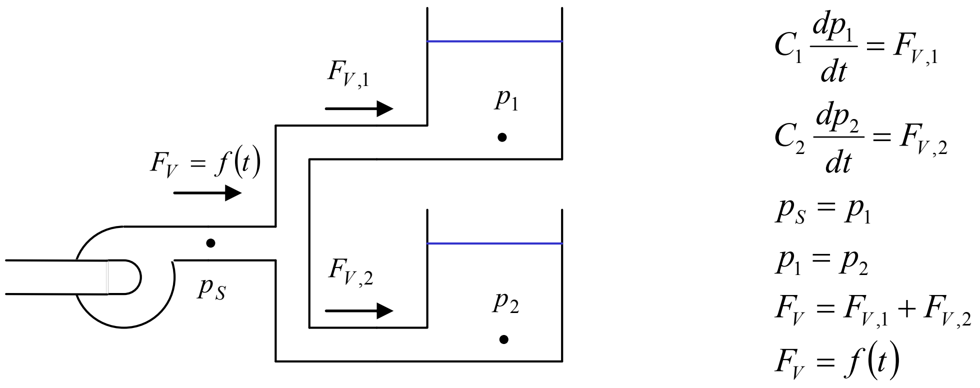

$F_{V,1}$
p_1
$F_V = f(t)$
p_S
$F_{V,2}$
p_2
$C_1 \frac{dp_1}{dt} = F_{V,1}$
$C_2 \frac{dp_2}{dt} = F_{V,2}$
$p_S = p_1$
$p_1 = p_2$
$F_V = F_{V,1} + F_{V,2}$
$F_V = f(t)$

Figure 5.1: Two tanks connected in parallel to a liquid source.

Let's analyze the computational causality of the model. Firstly, we replace the derivatives by dummy variables:

$$\frac{dp_1}{dt} \to derp_1 \qquad\qquad \frac{dp_2}{dt} \to derp_2 \tag{5.2}$$

The model shown in Figure 5.1, with the derivatives replaced by dummy variables, is the following:

$$C_1 \cdot derp_1 = F_{V,1} \tag{5.3}$$
$$C_2 \cdot derp_2 = F_{V,2} \tag{5.4}$$
$$p_S = p_1 \tag{5.5}$$
$$p_1 = p_2 \tag{5.6}$$
$$F_V = F_{V,1} + F_{V,2} \tag{5.7}$$
$$F_V = f(t) \tag{5.8}$$

Assuming that the two variables that appear differentiated (p_1 and p_2) can be selected as state variables, the model variables are classified as follows:

– Known:	t
	C_1, C_2
	p_1, p_2
– Unknown:	$F_{V,1}$, $F_{V,2}$, F_V, p_S
	$derp_1$, $derp_2$

The original incidence matrix is shown below.

$$
\begin{array}{c} \\ C_1 \cdot derp_1 = F_{V,1} \\ C_2 \cdot derp_2 = F_{V,2} \\ p_S = p_1 \\ p_1 = p_2 \\ F_V = F_{V,1} + F_{V,2} \\ F_V = f(t) \end{array}
\begin{array}{c} \begin{array}{cccccc} F_{V,1} & F_{V,2} & F_V & p_S & derp_1 & derp_2 \end{array} \\
\left(\begin{array}{cccccc}
X & 0 & 0 & 0 & X & 0 \\
0 & X & 0 & 0 & 0 & X \\
0 & 0 & 0 & X & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
X & X & X & 0 & 0 & 0 \\
0 & 0 & X & 0 & 0 & 0
\end{array}\right) \end{array}
\tag{5.9}
$$

As all the elements of the fourth row are zero, the model is structurally singular. Let's analyze the computational causality applying the rules of the partition algorithm.

1. Eq. (5.5) contains one unknown variable: p_S. Eq. (5.8) contains one unknown variable: F_V. These variables have to be evaluated from these equations. The variables are moved to the first columns and the equations to the first rows.

   $derp_1$ only appears in Eq. (5.3). $derp_2$ only appears in Eq. (5.4). Therefore, these variables have to be calculated from these equations. As the values of $derp_1$ and $derp_2$ are not employed in calculating other unknown variables ($derp_1$ and $derp_2$ only intervene in one equation), the variables are moved to the last columns, and the equations Eqs. (5.3) and (5.4) to the last rows.

$$
\begin{array}{c} \\ p_S = p_1 \\ F_V = f(t) \\ p_1 = p_2 \\ F_V = F_{V,1} + F_{V,2} \\ C_1 \cdot derp_1 = F_{V,1} \\ C_2 \cdot derp_2 = F_{V,2} \end{array}
\begin{array}{c} \begin{array}{cccccc} p_S & F_V & F_{V,1} & F_{V,2} & derp_1 & derp_2 \end{array} \\
\left(\begin{array}{cccccc}
\boxed{X} & 0 & 0 & 0 & 0 & 0 \\
0 & \boxed{X} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & X & X & X & 0 & 0 \\
0 & 0 & X & 0 & \boxed{X} & 0 \\
0 & 0 & 0 & X & 0 & \boxed{X}
\end{array}\right) \end{array}
\tag{5.10}
$$

2. There are two equations, Eqs. (5.6) and (5.7), and two unknown variables, $F_{V,1}$ and $F_{V,2}$, without annotated computational causality. However, the partition algorithm cannot proceed: Eq. (5.6) does not contain any unknown variable, and there is only one equation, Eq. (5.7), to calculate two unknown variables ($F_{V,1}$ and $F_{V,2}$).

The partition algorithm fails in this case because we have supposed that the model has two DoF, and we have selected p_1 and p_2 as state variables, but the two-tank model does not have two DoF, but only one.

The model of each tank, considered individually, has one DoF. Its initial state is set by specifying the amount of liquid stored initially, for instance, giving an initial value to the bottom pressure.

However, connecting the two tanks imposes their bottom pressures to be equal ($p_1 = p_2$), and this constraint reduces the number of DoF. The initial value of one pressure determines the initial value of the other pressure.

As the two-tank model has only 1 DoF, let's select as state variable only one of the pressures, for instance p_1. In this way, p_2 is not state variable. It is classified as unknown variable and has to be calculated from the model variables. The model with the computational causality annotated is shown below.

$$\begin{aligned}
&p_1 && \text{state variable} && (5.11)\\
&[F_V] &=\;& f(t) && (5.12)\\
&[p_S] &=\;& p_1 && (5.13)\\
&p_1 &=\;& [p_2] && (5.14)\\
&C_2 \cdot \frac{dp_2}{dt} &=\;& [F_{V,2}] \;\leftarrow\; \text{numerical differentiation} && (5.15)\\
&F_V &=\;& [F_{V,1}] + F_{V,2} && (5.16)\\
&C_1 \cdot [derp_1] &=\;& F_{V,1} && (5.17)
\end{aligned}$$

This computational causality implies numerical differentiation and, for this reason, it is not a good method from the numerical solution standpoint. Let's explore a different approach.

Eq. (5.6), this is, $p_1 = p_2$, is the constraint that reduces the number of DoF. A solution could be to **replace** this equation by its derivative (over-dot notation is used to represent derivative with respect to time):

$$p_1 = p_2 \qquad \rightarrow \qquad \dot{p}_1 = \dot{p}_2 \tag{5.18}$$

In this way, the model has 2 DoF and both pressures can be selected as state variables.

$$C_1 \cdot \dot{p}_1 = F_{V,1} \tag{5.19}$$

$$C_2 \cdot \dot{p}_2 = F_{V,2} \tag{5.20}$$

$$
\begin{aligned}
p_S &= p_1 && && (5.21)\\
\dot{p}_1 &= \dot{p}_2 && \leftarrow \text{ derivative of the constraint} && (5.22)\\
F_V &= F_{V,1} + F_{V,2} && && (5.23)\\
F_V &= f\,(t) && && (5.24)
\end{aligned}
$$

For this model to have the same solution as the original one, the initial values of the pressures must satisfy the original constraint: the initial value of p_1 must be equal to the initial value of p_2. The model, with the computational causality annotated, is shown below.

$$
\begin{aligned}
p_1, p_2 & \quad \text{state variables} && (5.25)\\
[F_V] &= f\,(t) && (5.26)\\
[p_S] &= p_1 && (5.27)\\
C_1 \cdot derp_1 &= F_{V,1} && (5.28)\\
C_2 \cdot derp_2 &= F_{V,2} && (5.29)\\
F_V &= F_{V,1} + F_{V,2} && (5.30)\\
derp_1 &= derp_2 && (5.31)
\end{aligned}
$$

where $derp_1$, $derp_2$, $F_{V,1}$ and $F_{V,2}$ are calculated by solving the system of simultaneous equations composed of the four last equations.

By reasoning as previously, we are assuming that $p_1(t) = p_2(t)$ is equivalent to impose both the equality of the initial values, $p_1(t_0) = p_2(t_0)$, and the equality of the derivatives, $\dot{p}_1(t) = \dot{p}_2(t)$.

This assumption is mathematically correct, but it can be lead to erroneous results if the model is solved numerically. If the numerical integrations of $derp_1$ and $derp_2$ are performed separately, the obtained values of p_1 and p_2 can be slightly different, due to the numerical errors. This difference in the values of p_1 and p_2, which is a numerical artifice that does not correspond to the behavior described by the original model, can be non-negligible in some applications of the model. In consequence, the method consisting in replacing the constraint by its derivative is not adequate for our purposes.

Let's explore another approach. It consists in **adding** to the model (instead of replacing) the derivative of the constraint that reduces the number of DoF, this is, including the equation $\dot{p}_1 = \dot{p}_2$ in the model; and selecting only one of the pressures as state variable, for instance, p_1. In this way, the model has 7 equations and 7 unknown variables: p_2, $F_{V,1}$, $F_{V,2}$, F_V, p_S, $derp_1$, $derp_2$. The model, with the computational causality annotated, is shown below.

$$\begin{aligned}
&p_1 && \text{state variable} && (5.32)\\
&[F_V] = f(t) && && (5.33)\\
&[p_S] = p_1 && && (5.34)\\
&p_1 = [p_2] && \leftarrow \text{ constraint} && (5.35)\\
&C_1 \cdot derp_1 = F_{V,1} && && (5.36)\\
&C_2 \cdot derp_2 = F_{V,2} && && (5.37)\\
&derp_1 = derp_2 && \leftarrow \text{ derivative of the constraint} && (5.38)\\
&F_V = F_{V,1} + F_{V,2} && && (5.39)
\end{aligned}$$

where $derp_1$, $derp_2$, $F_{V,1}$ and $F_{V,2}$ are calculated by solving the system of simultaneous equations formed by the last four equations. This approach produces a satisfactory result. In fact, this is the procedure employed by Modelica modeling environments such as Dymola and OpenModelica. The **procedure for simulating structurally singular DAE systems** is based on the following principles:

- The time derivatives are replaced by dummy variables. These dummy variables are classified as unknown variables for the assignment of computational causality and have to be calculated from the model equations.
- Certain model equations are differentiated symbolically a certain number of times, and these differentiated equations are added to the model. No equation is removed from the model.
- The number of variables selected as state variables is equal to the number of DoF of the model.
- The variables selected as state variables are calculated by numerical integration of their derivatives. Therefore, the state variables are classified as known variables for the assignment of computational causality.
- The variables that appear differentiated in the model and have not been selected as state variables are classified as unknown variables for the assignment of computational causality. These variables have to be calculated from the model equations.

The following question arises: is there a criteria to decide which model equations to differentiate, and how many times? Essentially, the answer to this question is: equations are differentiated if the new equations obtained by differentiating provide additional useful information. Another question arises: is there a criteria to calculate the number of DoF of a DAE system?

There are algorithms that automatically provide answers to these questions. One of these algorithms is the **Pantelides algorithm**. By analyzing the computational structure of the model, the Pantelides algorithm allows to determine which model equations have to be differentiated and how many times.

Dymola and OpenModelica implement variants of the Pantelides algorithm, symbolic formula manipulation and simplification algorithms, and algorithms for selecting the state variables. As a result, the complete process is performed automatically. An example is shown below.

The model of the system depicted in Figure 5.1 is described in Modelica Code 5.1. Before translating the model, we can ask Dymola to **write the differentiated equations in the log window**. This is accomplished by selecting *Output information when differentiating for index reduction* in the *Simulation Setup* window (see the option (3) in Figure 5.2). In this way, during the translation of Modelica Code 5.1, Dymola shows the following message in the log window:

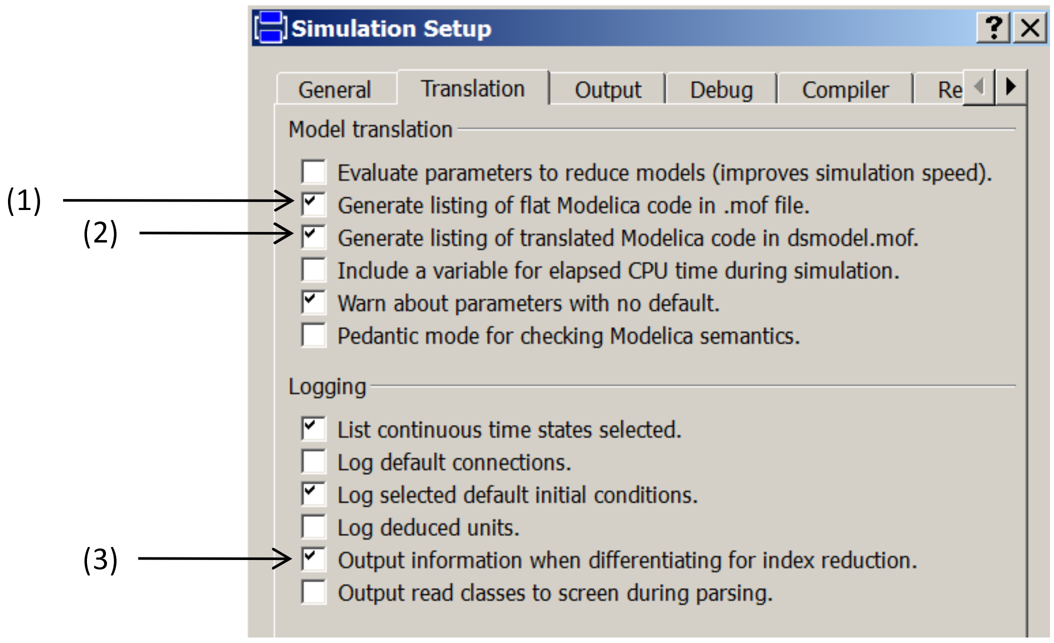
```
Differentiated the equation
p2 = p1;

giving
der(p2) = der(p1);

Selected continuous time states
   Statically selected continuous time states
   p1
```

Dymola has differentiated symbolically the $p_2 = p_1$ equation, has included in the model the obtained equation ($der(p_2) = der(p_1)$), and has selected p_1 as state variable. The initial value of the state variables can be modified in the *Variable Browser* window. As shown in Figure 5.3, Dymola allows to modify the initial value of p_1.

Selecting before the model translation the options signaled as (1) and (2) in Figure 5.2, Dymola **saves to file the flat model**, and the **sorted and solved model**. These are saved in the working directory to files named *TwoTanks.mof* and *dsmodel.mof*, respectively.

```
model TwoTanks
  import SI = Modelica.SIunits;
  SI.Pressure p1;
  SI.Pressure p2;
  SI.Pressure pS;
  SI.VolumeFlowRate Fv;
  SI.VolumeFlowRate Fv1;
  SI.VolumeFlowRate Fv2;
  parameter Real C1(unit="m3/Pa") = 1e-5;
  parameter Real C2(unit="m3/Pa") = 3e-5;
equation
  C1*der(p1) = Fv1;
  C2*der(p2) = Fv2;
  pS = p1;
  p1 = p2;
  Fv = Fv1 + Fv2;
  Fv = sin(time);
end TwoTanks;
```

Modelica Code 5.1: Two-tank and source system shown in Figure 5.1.



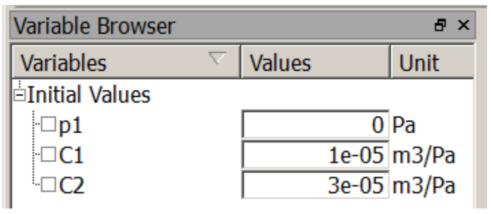
Model translation

- [ ] Evaluate parameters to reduce models (improves simulation speed).
- [✓] Generate listing of flat Modelica code in .mof file.
- [✓] Generate listing of translated Modelica code in dsmodel.mof.
- [ ] Include a variable for elapsed CPU time during simulation.
- [✓] Warn about parameters with no default.
- [ ] Pedantic mode for checking Modelica semantics.

Logging

- [✓] List continuous time states selected.
- [ ] Log default connections.
- [✓] Log selected default initial conditions.
- [ ] Log deduced units.
- [✓] Output information when differentiating for index reduction.
- [ ] Output read classes to screen during parsing.

Figure 5.2: *Simulation Setup* window of Dymola. (1): save to text file the flat model; (2): save to text file the solved and sorted model; and (3): show the differentiated equations in the log window.



Variables	Values	Unit
Initial Values		
p1	0	Pa
C1	1e-05	m3/Pa
C2	3e-05	m3/Pa

Figure 5.3: Model initialization.

As will be explained in Section 5.5, Modelica allows the model developer to **select the state variables**. This is accomplished through the *stateSelect* attribute of Real variables. If the *StateSelect.always* value is assigned to this attribute, the modeling environment selects this variable as state variable. For instance, by declaring the p_2 variable as shown below, it is selected as state variable.

```
SI.Pressure p2 ( stateSelect = StateSelect.always );
```

5.3 Index of DAE systems

The **index** is a property of DAE systems that is related to their computational causality. DAE systems with index larger than one are called **high-index DAE systems**, and exhibit the following property: the assignment of computational causality has not any solution that allows to select as state variables all the variables that appear differentiated.

As shown in the previous section, this type of systems arises, for instance, when the component connections reduce the number of DoF. In other words, when the DoF of the complete system is smaller than the sum of the DoF of the components taken separately.

The concept of index will be defined in this section, and the difficulties associated to the numerical solution of high-index DAE systems will be explained. These difficulties are the reason why Modelica modeling environments such as Dymola and OpenModelica manipulate symbolically the high-index DAE systems, performing index reduction before computing the numerical solution.

5.3.1 Definition of index

The **index** of a DAE system

$$\mathbf{F}(t, \mathbf{x}, \dot{\mathbf{x}}) = \mathbf{0} \tag{5.40}$$

is defined as

> the minimum number of times that all or part of (5.40) must be differentiated with respect to t in order to determine $\dot{\mathbf{x}}$ as a continuous function of $\mathbf{x}$, t.

The following examples illustrate index calculations.

Example 1. The explicit system of ordinary differential equations (ODE) shown in Eq. (5.41) has **index-0**, because $\dot{\mathbf{x}}$ is already written as a continuous function of $\mathbf{x}$, t.

$$\dot{\mathbf{x}} = \mathbf{F}\left(t, \mathbf{x}\right) \tag{5.41}$$

Example 2. The semi-explicit DAE shown in Eqs. (5.42) – (5.43) has **index-1** if and only if the Jacobian matrix of $\mathbf{F}_2$ with respect to $\mathbf{x}_2$ is non-singular, this is, the Jacobian matrix has a non-zero determinant and, consequently, has an inverse.

$$\dot{\mathbf{x}}_1 = \mathbf{F}_1\left(\mathbf{x}_1, \mathbf{x}_2, t\right) \tag{5.42}$$

$$0 = \mathbf{F}_2\left(\mathbf{x}_1, \mathbf{x}_2, t\right) \tag{5.43}$$

Observe that differentiating with respect to t the algebraic equations of the DAE system, this is, Eqs. (5.43), it is obtained:

$$0 = \frac{\partial \mathbf{F}_2\left(\mathbf{x}_1, \mathbf{x}_2, t\right)}{\partial \mathbf{x}_1} \cdot \dot{\mathbf{x}}_1 + \frac{\partial \mathbf{F}_2\left(\mathbf{x}_1, \mathbf{x}_2, t\right)}{\partial \mathbf{x}_2} \cdot \dot{\mathbf{x}}_2 + \frac{\partial \mathbf{F}_2\left(\mathbf{x}_1, \mathbf{x}_2, t\right)}{\partial t} \tag{5.44}$$

If $\frac{\partial \mathbf{F}_2}{\partial \mathbf{x}_2}$ is non-singular, then $\dot{\mathbf{x}}_2$ can be calculated from Eq. (5.44). Replacing $\dot{\mathbf{x}}_1$ with $\mathbf{F}_1\left(\mathbf{x}_1, \mathbf{x}_2, t\right)$, then $\dot{\mathbf{x}}_2$ is obtained as a continuous function of $\mathbf{x}_1$, $\mathbf{x}_2$, t.

Example 3. As a particular case of Example 2, let's consider the following DAE system

$$\dot{x} = x + y \tag{5.45}$$

$$0 = x + 2 \cdot y + a(t) \tag{5.46}$$

where $a\left(t\right)$ is a continuously differentiable function of t. Differentiating Eq. (5.46) with respect to t, and solving $\dot{y}$ from the obtained equation, it is obtained

$$\dot{x} = x + y \tag{5.47}$$

$$\dot{y} = -\frac{\dot{x} + \dot{a}(t)}{2} \tag{5.48}$$

and replacing Eq. (5.47) in (5.48), the explicit ODE shown below is obtained. Therefore, the DAE system shown in Eqs. (5.45) and (5.46) has **index-1**.

$$\dot{x} = x + y \tag{5.49}$$
$$\dot{y} = -\frac{x + y + \dot{a}(t)}{2} \tag{5.50}$$

Example 4. The DAE system shown in Eqs. (5.51) – (5.52) has **index-2** if and only if the Jacobian matrix of $\mathbf{F}_1$ with respect to $\mathbf{x}_2$, and the Jacobian matrix of $\mathbf{F}_2$ with respect to $\mathbf{x}_1$ are non-singular. Observe that, in order to obtain the system as an explicit ODE, Eq. (5.51) has to be differentiated once and Eq. (5.52) has to be differentiated twice.

$$\dot{\mathbf{x}}_1 = \mathbf{F}_1(\mathbf{x}_1, \mathbf{x}_2, t) \tag{5.51}$$
$$0 = \mathbf{F}_2(\mathbf{x}_1, t) \tag{5.52}$$

Example 5. Let's consider the DAE system composed by Eqs. (5.53) and (5.54).

$$\dot{x}_1 = x_2 \tag{5.53}$$
$$x_1 = t^2 + t + 2 \tag{5.54}$$

The second equation contains only one variable: x_1. Therefore, the derivative of x_1 has to be obtained from this equation, and the derivative of x_2 has to be obtained from the first equation. Differentiating the system, it is obtained:

$$\ddot{x}_1 = \dot{x}_2 \tag{5.55}$$
$$\dot{x}_1 = 2 \cdot t + 1 \tag{5.56}$$

Differentiating the second equation, it is obtained:

$$\ddot{x}_1 = 2 \tag{5.57}$$

Replacing (5.57) in (5.55), the system is written as an explicit ODE:

$$\begin{aligned} \dot{x}_1 &= 2 \cdot t + 1 && (5.58) \\ \dot{x}_2 &= 2 && (5.59) \end{aligned}$$

As we needed to differentiate twice, the DAE system composed by Eqs. (5.53) and (5.54) has **index-2**.

Example 6. Let's consider the following DAE system:

$$\begin{aligned} \dot{x}_1 &= f_1(t) - x_3 && (5.60) \\ \dot{x}_2 &= f_2(t) - x_1 && (5.61) \\ x_2 &= f_3(t) && (5.62) \end{aligned}$$

As x_3 only appears in Eq. (5.60), it is necessary to differentiate this equation to calculate $\dot{x}_3$. On the other hand, Eq. (5.62) contains only one variable, x_2, and consequently it is necessary to differentiate this equation to calculate $\dot{x}_2$. Finally, $\dot{x}_1$ is obtained by differentiating Eq. (5.61). Differentiating Eqs. (5.60) – (5.62), it is obtained:

$$\begin{aligned} \ddot{x}_1 &= \dot{f}_1(t) - \dot{x}_3 && (5.63) \\ \ddot{x}_2 &= \dot{f}_2(t) - \dot{x}_1 && (5.64) \\ \dot{x}_2 &= \dot{f}_3(t) && (5.65) \end{aligned}$$

Observe that $\ddot{x}_1$ and $\ddot{x}_2$ appear in Eqs. (5.63) and (5.64), respectively.

To obtain these second-order derivatives as a function of the variables (x_1, x_2, x_3), the first-order derivatives ($\dot{x}_1$, $\dot{x}_2$, $\dot{x}_3$) and time (t), Eq. (5.64) is differentiated:

$$\dddot{x}_2 = \ddot{f}_2(t) - \ddot{x}_1 \qquad (5.66)$$

The second-order and third-order derivatives of x_2 are obtained differentiating twice Eq. (5.65):

$$\ddot{x}_2 = \ddot{f}_3(t) \tag{5.67}$$
$$\dddot{x}_2 = \dddot{f}_3(t) \tag{5.68}$$

Replacing, it is obtained:

$$\ddot{f}_2(t) - \dddot{f}_3(t) = \dot{f}_1(t) - \dot{x}_3 \tag{5.69}$$
$$\ddot{f}_3(t) = \dot{f}_2(t) - \dot{x}_1 \tag{5.70}$$
$$\dot{x}_2 = \dot{f}_3(t) \tag{5.71}$$

Manipulating these equations, the ODE shown below is obtained. As it has been necessary to differentiate three times, the DAE system has **index-3**.

$$\dot{x}_1 = \dot{f}_2(t) - \ddot{f}_3(t) \tag{5.72}$$
$$\dot{x}_2 = \dot{f}_3(t) \tag{5.73}$$
$$\dot{x}_3 = \dot{f}_1(t) - \ddot{f}_2(t) + \dddot{f}_3(t) \tag{5.74}$$

Example 7. The following property may be useful for calculating the index of DAE systems. If the DAE system

$$\mathbf{F}\left(t, \mathbf{x}, \dot{\mathbf{x}}\right) = 0 \tag{5.75}$$

has **index-n**, then the semi-explicit DAE

$$\dot{\mathbf{x}} = \mathbf{z} \tag{5.76}$$
$$\mathbf{F}\left(t, \mathbf{x}, \mathbf{z}\right) = 0 \tag{5.77}$$

has **index-(n+1)**.

If the DAE shown in Eq. (5.75) has index-n, then $\dot{\mathbf{x}}$ is obtained as a function of $\mathbf{x}$, t by differentiating n times part or all the system. On the other hand, as described by Eq. (5.76), $\dot{\mathbf{x}}$ is equal to $\mathbf{z}$. In consequence, $\mathbf{z}$ is obtained as a function of $\mathbf{x}$, t by differentiating n times part or all the system. Therefore, $\dot{\mathbf{z}}$ is obtained by differentiating one more time.

Example 8. Let's revisit the two-tank model described in Section 5.2. The model equations are as follows.

$$C_1 \cdot \dot{p}_1 = F_{V,1} \tag{5.78}$$
$$C_2 \cdot \dot{p}_2 = F_{V,2} \tag{5.79}$$
$$p_S = p_1 \tag{5.80}$$
$$p_1 = p_2 \tag{5.81}$$
$$F_V = F_{V,1} + F_{V,2} \tag{5.82}$$
$$F_V = f(t) \tag{5.83}$$

In order to calculate the index of the DAE system composed of Eqs. (5.78) – (5.83), we analyze what equations have to be differentiated to obtain an explicit ODE

$$\dot{\mathbf{x}} = \mathbf{g}(\mathbf{x}, t) \tag{5.84}$$

where

$$\mathbf{x} = \{p_1, p_2, p_S, F_V, F_{V,1}, F_{V,2}\} \tag{5.85}$$

Differentiating the algebraic equations of the DAE system, it is obtained:

$$C_1 \cdot \dot{p}_1 = F_{V,1} \tag{5.86}$$
$$C_2 \cdot \dot{p}_2 = F_{V,2} \tag{5.87}$$
$$\dot{p}_S = \dot{p}_1 \tag{5.88}$$
$$\dot{p}_1 = \dot{p}_2 \tag{5.89}$$
$$\dot{F}_V = \dot{F}_{V,1} + \dot{F}_{V,2} \tag{5.90}$$
$$\dot{F}_V = \dot{f}(t) \tag{5.91}$$

Let's analyze whether $\dot{\mathbf{x}}$ can be obtained from Eqs. (5.86) – (5.91). To this end, we calculate the computational causality of these equations, assuming that the unknown variables are $\dot{\mathbf{x}}$. The original incidence matrix is shown below.

$$
\begin{array}{r|cccccc|}
 & \dot{p}_1 & \dot{p}_2 & \dot{F}_V & \dot{p}_S & \dot{F}_{V,1} & \dot{F}_{V,2} \\
C_1 \cdot \dot{p}_1 = F_{V,1} & X & 0 & 0 & 0 & 0 & 0 \\
C_2 \cdot \dot{p}_2 = F_{V,2} & 0 & X & 0 & 0 & 0 & 0 \\
\dot{p}_S = \dot{p}_1 & X & 0 & 0 & X & 0 & 0 \\
\dot{p}_1 = \dot{p}_2 & X & X & 0 & 0 & 0 & 0 \\
\dot{F}_V = \dot{F}_{V,1} + \dot{F}_{V,2} & 0 & 0 & X & 0 & X & X \\
\dot{F}_V = \dot{f}(t) & 0 & 0 & X & 0 & 0 & 0
\end{array}
\tag{5.92}
$$

1. There are three equations that contain only one unknown variable. These are: Eq. (5.86) – $\dot{p}_1$; Eq. (5.87) – $\dot{p}_2$; and Eq. (5.91) – $\dot{F}_V$. Therefore, these equations have to be employed to calculate these unknown variables. These equations are moved to the firsts rows of the matrix and the variables to the firsts columns.

   The $\dot{p}_S$ variable only appears in one equation: Eq. (5.88). Therefore, this variable has to be calculated from this equation. As this variable does not intervene in any other equation, it is moved to the last column and Eq. (5.88) is moved to the last row.

$$
\begin{array}{r|cccccc|}
 & \dot{p}_1 & \dot{p}_2 & \dot{F}_V & \dot{F}_{V,1} & \dot{F}_{V,2} & \dot{p}_S \\
C_1 \cdot \dot{p}_1 = F_{V,1} & \boxed{X} & 0 & 0 & 0 & 0 & 0 \\
C_2 \cdot \dot{p}_2 = F_{V,2} & 0 & \boxed{X} & 0 & 0 & 0 & 0 \\
\dot{F}_V = \dot{f}(t) & 0 & 0 & \boxed{X} & 0 & 0 & 0 \\
\dot{p}_1 = \dot{p}_2 & X & X & 0 & 0 & 0 & 0 \\
\dot{F}_V = \dot{F}_{V,1} + \dot{F}_{V,2} & 0 & 0 & X & X & X & 0 \\
\dot{p}_S = \dot{p}_1 & X & 0 & 0 & 0 & 0 & \boxed{X}
\end{array}
\tag{5.93}
$$

2. Eq. (5.89) is redundant, given that $\dot{p}_1$ and $\dot{p}_2$ are evaluated from other equations. On the other hand, there is only one equation, Eq. (5.90), to calculate two unknown variables: $\dot{F}_{V,1}$ and $\dot{F}_{V,2}$.

The conclusion is that Eqs. (5.86) – (5.91) are not a well-defined ODE system. We proceed differentiating. The equations to be differentiated are selected attending to the following reasoning.

– $\dot{p}_S$ appears in only one equation: Eq. (5.88). This is the reason because Eq. (5.88) has to be employed for evaluating $\dot{p}_S$. If Eq. (5.88) is differentiated, a new variable is introduced, $\ddot{p}_S$, which has to be evaluated from this new equation. As differentiation of Eq. (5.88) does not impose an additional constraint on the unknown variables, Eq. (5.88) is not selected for differentiation.

- The same reasoning applies to Eqs. (5.90) and (5.91), whose differentiation introduces new unknown variables: $\ddot{F}_V$, $\ddot{F}_{V,1}$ and $\ddot{F}_{V,2}$.
- In consequence, we select to differentiate Eqs. (5.86), (5.87) and (5.89). It is obtained:

$$\begin{aligned} C_1 \cdot \ddot{p}_1 &= \dot{F}_{V,1} && (5.94)\\ C_2 \cdot \ddot{p}_2 &= \dot{F}_{V,2} && (5.95)\\ \ddot{p}_1 &= \ddot{p}_2 && (5.96) \end{aligned}$$

Adding these three equations to the system, and removing from the system the redundant equation (5.89), it is obtained:

$$\begin{aligned} C_1 \cdot \dot{p}_1 &= F_{V,1} && (5.97)\\ C_2 \cdot \dot{p}_2 &= F_{V,2} && (5.98)\\ \dot{p}_S &= \dot{p}_1 && (5.99)\\ \dot{F}_V &= \dot{F}_{V,1} + \dot{F}_{V,2} && (5.100)\\ \dot{F}_V &= \dot{f}\,(t) && (5.101)\\ C_1 \cdot \ddot{p}_1 &= \dot{F}_{V,1} && (5.102)\\ C_2 \cdot \ddot{p}_2 &= \dot{F}_{V,2} && (5.103)\\ \ddot{p}_1 &= \ddot{p}_2 && (5.104) \end{aligned}$$

Considering that the unknown variables are $\{\dot{p}_1, \dot{p}_2, \dot{p}_S, \dot{F}_V, \dot{F}_{V,1}, \dot{F}_{V,2}, \ddot{p}_1, \ddot{p}_2\}$, the computational causality can be assigned as described below.

- Eq. (5.97) contains only one unknown variable: $\dot{p}_1$. Therefore, this equation has to be employed to evaluate $\dot{p}_1$. For the same reason, $\dot{p}_2$ has to be evaluated from Eq. (5.98) and $\dot{F}_V$ from Eq. (5.101).
- $\dot{p}_S$ has to be evaluated from Eq. (5.99), where $\dot{p}_1$ can be replaced with $\frac{F_{V,1}}{C_1}$.
- Finally, $\dot{F}_{V,1}$, $\dot{F}_{V,2}$, $\ddot{p}_1$ and $\ddot{p}_2$ can be calculated from solving the following linear system of simultaneous equations

$$\dot{F}_V = \dot{F}_{V,1} + \dot{F}_{V,2} \tag{5.105}$$
$$C_1 \cdot \ddot{p}_1 = \dot{F}_{V,1} \tag{5.106}$$
$$C_2 \cdot \ddot{p}_2 = \dot{F}_{V,2} \tag{5.107}$$
$$\ddot{p}_1 = \ddot{p}_2 \tag{5.108}$$

The BLT incidence matrix is shown below.

$$\begin{array}{c|cccccccc}
 & \dot{p}_1 & \dot{p}_2 & \dot{F}_V & \dot{p}_S & \dot{F}_{V,1} & \dot{F}_{V,2} & \ddot{p}_1 & \ddot{p}_2 \\
\hline
C_1 \cdot \dot{p}_1 = F_{V,1} & \boxed{X} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
C_2 \cdot \dot{p}_2 = F_{V,2} & 0 & \boxed{X} & 0 & 0 & 0 & 0 & 0 & 0 \\
\dot{F}_V = \dot{f}(t) & 0 & 0 & \boxed{X} & 0 & 0 & 0 & 0 & 0 \\
\dot{p}_S = \dot{p}_1 & X & 0 & 0 & \boxed{X} & 0 & 0 & 0 & 0 \\
\dot{F}_V = \dot{F}_{V,1} + \dot{F}_{V,2} & 0 & 0 & X & 0 & X & X & 0 & 0 \\
C_1 \cdot \ddot{p}_1 = \dot{F}_{V,1} & 0 & 0 & 0 & 0 & X & 0 & X & 0 \\
C_2 \cdot \ddot{p}_2 = \dot{F}_{V,2} & 0 & 0 & 0 & 0 & 0 & X & 0 & X \\
\ddot{p}_1 = \ddot{p}_2 & 0 & 0 & 0 & 0 & 0 & 0 & X & X
\end{array} \tag{5.109}$$

The expressions to calculate $\dot{F}_{V,1}$ and $\dot{F}_{V,2}$ can be obtained by eliminating $\ddot{p}_1$ and $\ddot{p}_2$ in Eqs. (5.105) – (5.108)

$$\dot{F}_{V,1} = C_1 \cdot \frac{\dot{F}_V}{C_1 + C_2} \tag{5.110}$$
$$\dot{F}_{V,2} = C_2 \cdot \frac{\dot{F}_V}{C_1 + C_2} \tag{5.111}$$

and by replacing $\dot{F}_V$ with $\dot{f}\,(t)$,

$$\dot{F}_{V,1} = C_1 \cdot \frac{\dot{f}\,(t)}{C_1 + C_2} \tag{5.112}$$
$$\dot{F}_{V,2} = C_2 \cdot \frac{\dot{f}\,(t)}{C_1 + C_2} \tag{5.113}$$

The explicit ODE system shown below is obtained. In consequence, the original DAE system has **index-2**.

$$\dot{p}_1 = \frac{F_{V,1}}{C_1} \tag{5.114}$$

$$\dot{p}_2 = \frac{F_{V,2}}{C_2} \tag{5.115}$$

$$\dot{F}_V = \dot{f}\,(t) \tag{5.116}$$

$$\dot{p}_S = \frac{F_{V,1}}{C_1} \tag{5.117}$$

$$\dot{F}_{V,1} = C_1 \cdot \frac{\dot{f}\,(t)}{C_1 + C_2} \tag{5.118}$$

$$\dot{F}_{V,2} = C_2 \cdot \frac{\dot{f}\,(t)}{C_1 + C_2} \tag{5.119}$$

5.3.2 Difficulties in the numerical solution of high-index DAE systems

The index is not an intrinsic property of the modeled physical system, but a property of the equations employed to describe the model. In fact, we have seen that the index can be reduced by differentiating a number of times certain equations of the model.

Numerical algorithms work well for solving index-0 and index-1 DAE systems, and in some cases, also for solving index-2 DAE systems. However, they usually don't work well for DAE systems whose index is higher than 2. The reason is that the numerical solution of high-index (larger than one) DAE systems typically implies numerical differentiation. High-index DAE systems are symbolically manipulated by the Modelica modeling environments in order to reduce their index, before computing their numerical solution.

To illustrate this point, let's consider the following index-2 DAE system:

$$\dot{x}_1 = x_2 \tag{5.120}$$

$$x_1 = t^2 + t + 2 \tag{5.121}$$

Applying the implicit trapezoidal rule to the first equation, the following discretized system is obtained:

$$x_1(t_{k+1}) = x_1(t_k) + h \cdot \frac{x_2(t_{k+1}) + x_2(t_k)}{2} + \mathrm{O}\left(h^2\right) \tag{5.122}$$

$$x_1(t_{k+1}) = t_{k+1}^2 + t_{k+1} + 2 \tag{5.123}$$

where h is the time step of the integration method. The second equation can be employed to calculate $x_1(t_{k+1})$, and then the first equation can be employed to calculate $x_2(t_{k+1})$. Solving, it is obtained that $x_2(t_{k+1})$ is calculated by numerical differentiation, and the error term is not $\mathrm{O}(h^2)$, but $\mathrm{O}(h)$:

$$x_2(t_{k+1}) = 2 \cdot \underbrace{\frac{x_1(t_{k+1}) - x_1(t_k)}{h}}_{\text{Numerical differentiation}} - x_2(t_k) + \mathrm{O}(h) \tag{5.124}$$

The determination of the optimal step size is problematic in numerical differentiation: too small values of h produce large round-off error, while too large values of h produce large discretization error.

Let's apply a different integration method to the first equation, for instance, the explicit Euler method. The discretized system is shown below.

$$x_1(t_{k+1}) = x_1(t_k) + h \cdot x_2(t_k) \tag{5.125}$$

$$x_1(t_{k+1}) = t_{k+1}^2 + t_{k+1} + 2 \tag{5.126}$$

As $x_2(t_{k+1})$ does not appear in the equations, the system cannot be solved employing the explicit Euler method and, for the same reason, any explicit integration method.

On the other hand, let's consider the initialization of the DAE system (5.120) – (5.121). If x_1 can be selected as state variable, then an arbitrary initial value, $x_1(t_0)$, can be assigned to this variable at the initial time t_0. However, the initial value of x_1 must satisfy Eq. (5.121): the value of $x_1(t_0)$ is $t_0^2 + t_0 + 2$. Therefore, the initial value of x_1 cannot be assigned arbitrarily. The number of initial values that can be assigned arbitrarily in this system (i.e., the number of DoF) is zero, and the number of variables that appear differentiated is one. The number of DoF is smaller than the number of differentiated variables. This is a common property to high-index DAE systems.

5.4 Initialization of DAE systems

In order to analyze the initialization problem, let's represent the DAE system in the form shown in Eq. (5.127), where an explicit distinction is made between those variables that appear differentiated ($\mathbf{x} \in \Re^n$) and those that don't ($\mathbf{y} \in \Re^m$).

$$\mathbf{F}\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, t\right) = 0 \tag{5.127}$$

$\mathbf{F} : G \subseteq \Re^n \times \Re^n \times \Re^m \times \Re \to \Re^{n+m}$ are $n + m$ real functions that, in general, are non-linear with respect to $\dot{\mathbf{x}}$.

A necessary condition for $(\mathbf{x_0}, \dot{\mathbf{x}}_\mathbf{0}, \mathbf{y_0})$ to be a consistent set of initial conditions is that these initial values satisfy the **original DAE system** (5.127) at the initial time t_0. In other words, consistency requires that:

$$\mathbf{F}\left(\mathbf{x_0}, \dot{\mathbf{x}}_\mathbf{0}, \mathbf{y_0}, t_0\right) = 0 \tag{5.128}$$

However, satisfying the original system is not, in general, a sufficient condition. This is the case when additional constraints on the initial values $(\mathbf{x_0}, \dot{\mathbf{x}}_\mathbf{0}, \mathbf{y_0})$ are obtained from differentiating a number of times certain equations of the original DAE system. These equations, that impose additional constraints on the initial values and are obtained by differentiating a number of times certain equations of the original DAE system, are named **hidden constraints**.

There exist DAE systems that don't include hidden constraints. In these systems, differentiating the original equations introduces new variables, so that the new equations are satisfied by all possible values of $(\mathbf{x_0}, \dot{\mathbf{x}}_\mathbf{0}, \mathbf{y_0})$ and appropriate values of the new variables. In consequence, differentiation does not impose in these DAE systems additional constraints on the initial values $(\mathbf{x_0}, \dot{\mathbf{x}}_\mathbf{0}, \mathbf{y_0})$.

The existence of hidden constraints is related to the structural singularity of the original DAE system. Let's suppose that the computational causality of the original DAE system is assigned by assuming that all variables that appear differentiated are selected as state variables.

- If the original DAE system contains hidden constraints, then it is structurally singular. The objective of the symbolic differentiations performed by the Modelica modeling environments on structurally singular DAE systems is to obtain the hidden constraints and add them to the system. The index of the DAE system is reduced by adding the hidden constraints.
- If the original DAE system does not contain hidden constraints, then it is structurally non-singular and its BLT incidence matrix can be obtained.

5.4.1 Hidden constraints and index reduction

The examples shown in this section try to illustrate the search of hidden constraints, the index reduction and the assignment of computational causality. The applied procedure is as follows. Firstly, we analyze whether the original DAE system has hidden constraints. Then, we proceed in one of two ways.

– If the DAE system has not hidden constraints, then the computational causality is assigned selecting as state variables the variables that appear differentiated in the model.

– If the DAE system has hidden constraints:

1. The hidden constraints are added to the system. The resultant system, composed of the original equations and the hidden constraints, is named **extended DAE system**. The extended and original systems have the same mathematical solution and the same number of DoF, but the index of the extended DAE system is lower and the extended system is less difficult numerically.
2. The number of DoF of the extended DAE system is calculated. This number will be smaller than the number of variables that appear differentiated in the original DAE system.
3. The computational causality of the extended DAE system is assigned, selecting as many state variables as DoF has the DAE system.

Example 1. Consider again the DAE system discussed in the previous section.

$$\dot{x}_1 = x_2 \tag{5.129}$$
$$x_1 = t^2 + t + 2 \tag{5.130}$$

Let's analyze if the equations obtained by differentiating (5.129) and (5.130) impose additional constraints on the initial value of $(x_1, \dot{x}_1, x_2)$. Differentiating both equations, it is obtained:

$$\ddot{x}_1 = \dot{x}_2 \tag{5.131}$$
$$\dot{x}_1 = 2 \cdot t + 1 \tag{5.132}$$

As Eq. (5.131) contains only the $\ddot{x}_1$ and $\dot{x}_2$ variables, this equation does not impose a constraint on the initial value of $(x_1, \dot{x}_1, x_2)$.

Eq. (5.132) imposes a constraint on the initial value of $\dot{x}_1$, which must be equal to $2 \cdot t_0 + 1$.

The equations obtained by differentiating Eqs. (5.131) and (5.132) contain new variables, not imposing any additional constraint of the initial values of $(x_1, \dot{x}_1, x_2)$.

The extended DAE system is:

$$\dot{x}_1 = x_2 \tag{5.133}$$
$$x_1 = t^2 + t + 2 \tag{5.134}$$
$$\dot{x}_1 = 2 \cdot t + 1 \tag{5.135}$$

Let's name $(x_{1_0}, \dot{x}_{1_0}, x_{2_0})$ the initial value of $(x_1, \dot{x}_1, x_2)$, and t_0 the initial time. The following three equations have to be satisfied:

$$x_{1_0} = t_0^2 + t_0 + 2 \tag{5.136}$$
$$x_{2_0} = 2 \cdot t_0 + 1 \tag{5.137}$$
$$\dot{x}_{1_0} = 2 \cdot t_0 + 1 \tag{5.138}$$

The initial values are completely determined by the equations of the extended DAE system. The model has zero DoF, this is, zero state variables. Observe that the extended DAE system, composed of Eqs. (5.133) – (5.135), is equivalent to the original DAE system, and the numerical solution of the extended DAE system does not represent any difficulty. The extended model, sorted and solved, is shown below.

$$[x_1] = t^2 + t + 2 \tag{5.139}$$
$$[derx_1] = 2 \cdot t + 1 \tag{5.140}$$
$$[x_2] = derx_1 \tag{5.141}$$

Example 2. This example illustrates the case in which the differentiation of the DAE system equations does not bring up any additional constraint. Let's consider the following DAE system

$$\begin{aligned} \dot{x} &= x + y && (5.142) \\ 0 &= x + 2 \cdot y + a(t) && (5.143) \end{aligned}$$

where $a\,(t)$ is a continuously differentiable function of time. Differentiating these two equations with respect to time, the following two equations are obtained, which contain two new variables: $\dot{y}$ and $\ddot{x}$.

$$\begin{aligned} \ddot{x} &= \dot{x} + \dot{y} && (5.144) \\ 0 &= \dot{x} + 2 \cdot \dot{y} + \dot{a}(t) && (5.145) \end{aligned}$$

If the initial values of the new variables are selected as shown below, then these two new equations are satisfied by all initial values (x_0,$\dot{x}_0$,y_0) of the original variables $(x, \dot{x}, y)$. In consequence, differentiating the equations with respect to time does not produce additional constraints on the original variables.

$$\begin{aligned} \dot{y}_0 &= -\frac{\dot{x}_0 + \dot{a}(t_0)}{2} && (5.146) \\ \ddot{x}_0 &= \dot{x}_0 + \dot{y}_0 && (5.147) \end{aligned}$$

Differentiating again with respect to time, we are in the same situation. Successive differentiation introduces new variables: the successive derivatives of $\dot{x}$, y. In consequence, the equations obtained by differentiating successively don't introduce additional constraints on the original variables $(x, \dot{x}, y)$.

For this reason, the initial values (x_0,$\dot{x}_0$,y_0) only have to satisfy the original DAE system. The vector of initial values has three components, which have to satisfy two equations. Therefore, the DAE system has one DoF. One variable can be selected as state variable. Selecting x as state variable and assigning the computational causality, the following sorted and solved model is obtained:

$$\begin{aligned} [y] &= -\frac{x + a(t)}{2} && (5.148) \\ [derx] &= x + y && (5.149) \end{aligned}$$

Example 3. Let's consider the following DAE system

$$\dot{x}_1 + \dot{x}_2 = a(t) \tag{5.150}$$
$$x_1 + x_2^2 = b(t) \tag{5.151}$$

where $a(t)$ and $b(t)$ are continuously differentiable functions of time.

In order to analyze whether the initial conditions $(x_{1_0}, x_{2_0}, \dot{x}_{1_0}, \dot{x}_{2_0})$ have to satisfy additional constraints, let's differentiate the system with respect to time:

$$\ddot{x}_1 + \ddot{x}_2 = \dot{a}(t) \tag{5.152}$$
$$\dot{x}_1 + 2 \cdot x_2 \cdot \dot{x}_2 = \dot{b}(t) \tag{5.153}$$

The first equation introduces two new variables $(\ddot{x}_1, \ddot{x}_2)$ and does not contain the original variables $(x_1, x_2, \dot{x}_1, \dot{x}_2)$. The second equation is an additional constraint on the original variables. Therefore, the initial conditions $(x_{1_0}, x_{2_0}, \dot{x}_{1_0}, \dot{x}_{2_0})$ must satisfy:

$$\dot{x}_1 + \dot{x}_2 = a(t) \tag{5.154}$$
$$x_1 + x_2^2 = b(t) \tag{5.155}$$
$$\dot{x}_1 + 2 \cdot x_2 \cdot \dot{x}_2 = \dot{b}(t) \tag{5.156}$$

The second-order derivative of the original system is:

$$\frac{d^3x_1}{dt^3} + \frac{d^3x_2}{dt^3} = \ddot{a}(t) \tag{5.157}$$
$$\ddot{x}_1 + 2 \cdot \dot{x}_2^2 + 2 \cdot x_2\ddot{x}_2 = \ddot{b}(t) \tag{5.158}$$

The first equation introduces two new variables $(\frac{d^3x_1}{dt^3}, \frac{d^3x_2}{dt^3})$. The second equation contains $\dot{x}_2$ and x_2, but it does not impose an additional constraint if, for any initial value of $\dot{x}_2$ and x_2, it is possible to find initial values of $\ddot{x}_1$, $\ddot{x}_2$ that satisfy the system

$$\begin{aligned}\ddot{x}_1 + \ddot{x}_2 &= \dot{a}(t) &(5.159)\\ \ddot{x}_1 + 2\cdot\dot{x}_2^2 + 2\cdot x_2\cdot\ddot{x}_2 &= \ddot{b}(t) &(5.160)\end{aligned}$$

or equivalently

$$\begin{aligned}\ddot{x}_1 + \ddot{x}_2 &= \dot{a}(t) &(5.161)\\ \ddot{x}_1 + 2\cdot x_2\cdot\ddot{x}_2 &= \ddot{b}(t) - 2\cdot\dot{x}_2^2 &(5.162)\end{aligned}$$

The determinant of this 2×2 system of linear equations is $(2\cdot x_2 - 1)$. Therefore, if the initial value of x_2 is not equal to 0.5, then it is possible to find an initial value of $\ddot{x}_1$, $\ddot{x}_2$ that satisfies the system. Observe that if $x_{2_0} = 0.5$, then the left-hand expressions of the first and third equations

$$\begin{aligned}\dot{x}_1 + \dot{x}_2 &= a(t) &(5.163)\\ x_1 + x_2^2 &= b(t) &(5.164)\\ \dot{x}_1 + 2\cdot x_2\cdot\dot{x}_2 &= \dot{b}(t) &(5.165)\end{aligned}$$

are identical.

Differentiating again, it is obtained:

$$\begin{aligned}\frac{d^4x_1}{dt^4} + \frac{d^4x_2}{dt^4} &= \frac{d^3a(t)}{dt^3} &(5.166)\\ \frac{d^3x_1}{dt^3} + 4\cdot\dot{x}_2\cdot\ddot{x}_2 + 2\cdot\dot{x}_2\cdot\ddot{x}_2 + 2\cdot x_2\cdot\frac{d^3x_2}{dt^3} &= \frac{d^3b(t)}{dt^3} &(5.167)\end{aligned}$$

The first equation introduces two new variables $\left(\frac{d^4x_1}{dt^4}, \frac{d^4x_2}{dt^4}\right)$. The second equation contains $\dot{x}_2$ and $\ddot{x}_2$, but it does not represent an additional constraint. For any initial value of $\dot{x}_2$ and $\ddot{x}_2$, and assuming that $x_2 \neq 0.5$, it is possible to obtain an initial value of $\left(\frac{d^3x_1}{dt^3}, \frac{d^3x_2}{dt^3}\right)$ that satisfies the system

$$\frac{d^3x_1}{dt^3} + \frac{d^3x_2}{dt^3} = \ddot{a}(t) \tag{5.168}$$

$$\frac{d^3x_1}{dt^3} + 6 \cdot \dot{x}_2 \cdot \ddot{x}_2 + 2 \cdot x_2 \cdot \frac{d^3x_2}{dt^3} = \frac{d^3b(t)}{dt^3} \tag{5.169}$$

or equivalently,

$$\frac{d^3x_1}{dt^3} + \frac{d^3x_2}{dt^3} = \ddot{a}(t) \tag{5.170}$$

$$\frac{d^3x_1}{dt^3} + 2 \cdot x_2 \cdot \frac{d^3x_2}{dt^3} = \frac{d^3b(t)}{dt^3} - 6 \cdot \dot{x}_2 \cdot \ddot{x}_2 \tag{5.171}$$

And so on. Therefore, the initial value vector $(x_{1_0}, x_{2_0}, \dot{x}_{1_0}, \dot{x}_{2_0})$ must satisfy the following extended DAE system, which is mathematically equivalent to the original DAE system:

$$\dot{x}_1 + \dot{x}_2 = a(t) \tag{5.172}$$

$$x_1 + x_2^2 = b(t) \tag{5.173}$$

$$\dot{x}_1 + 2 \cdot x_2 \cdot \dot{x}_2 = \dot{b}(t) \tag{5.174}$$

As the initial value vector has four components and must satisfy three equations, the DAE system has one DoF. Selecting x_2 as state variable and assigning the computational causality, the following sorted and solved model is obtained:

$$[x_1] = b(t) - x_2^2 \tag{5.175}$$

$$[derx_2] = \frac{\dot{b}(t) - a(t)}{2 \cdot x_2 - 1} \tag{5.176}$$

$$[derx_1] = a(t) - derx_2 \tag{5.177}$$

5.4.2 The Pantelides algorithm

The Pantelides algorithm analyzes the structure of the DAE system to find the minimum subset of equations whose differentiation introduces additional constraints on the initial value vector. These constraints, together with the original equations, must be satisfied by the initial value vector.

The algorithm employs bipartite graphs for representing the computational structure of the DAE system, and it does not require performing arithmetic operations or symbolic differentiation.

The algorithm converges if the DAE system is well posed. The following two examples illustrate the algorithm application to a well-posed and an ill-posed DAE system. Both examples are based on the following model of a dynamical system:

$$\begin{aligned} 0 &= f_1(x, u_1, u_2) && (5.178)\\ 0 &= f_2(x, \dot{x}, y_1) && (5.179)\\ 0 &= f_3(x, y_2) && (5.180) \end{aligned}$$

where u_1 and u_2 are the manipulated variables of the system (inputs); y_1 and y_2 are the observed variables (outputs); and x is an internal variable that appears differentiated in the model. The objective is to simulate the system for the following two computational causalities (see Figure 5.4).

– *Direct problem*: the evolution of the input variables is known, and the objective is to calculate the evolution of the output variables.

– *Inverse problem*: the desired evolution of the output variables is known, and the objective is to calculate the inputs that have to be applied for obtaining these outputs.

Let's analyze the **direct problem** firstly. Assuming that u_1 and u_2 are known variables, and x is state variable, the computational causality is as follows

$$\begin{aligned} &x && \text{state variable} && && (5.181)\\ 0 &= f_1(x, u_1, u_2) && \leftarrow \text{redundant equation} && && (5.182)\\ 0 &= f_2(x, derx, y_1) && \leftarrow \text{unknown variables: } derx, y_1 && && (5.183)\\ 0 &= f_3(x, [y_2]) && && && (5.184) \end{aligned}$$

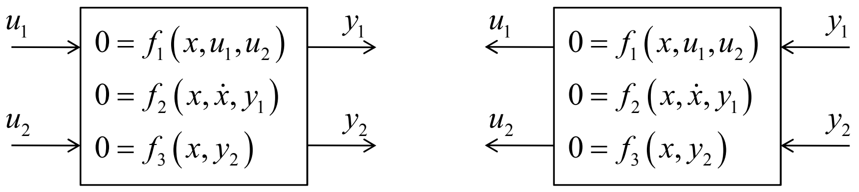


Figure 5.4: Computational causality of the direct (left) and inverse (right) problems.

This DAE system is structurally singular. Let's analyze whether the DAE system has high index or is mathematically incorrect. To this end, let's search for hidden constraints on the initial value of (x, $\dot{x}$, y_1, y_2) by differentiating the system equations. Differentiating with respect to time, it is obtained:

$$0 = \frac{\partial f_1}{\partial x} \cdot \dot{x} + \frac{\partial f_1}{\partial u_1} \cdot \dot{u}_1 + \frac{\partial f_1}{\partial u_2} \cdot \dot{u}_2 \tag{5.185}$$

$$0 = \frac{\partial f_2}{\partial x} \cdot \dot{x} + \frac{\partial f_2}{\partial \dot{x}} \cdot \ddot{x} + \frac{\partial f_2}{\partial y_1} \cdot \dot{y}_1 \tag{5.186}$$

$$0 = \frac{\partial f_3}{\partial x} \cdot \dot{x} + \frac{\partial f_3}{\partial y_2} \cdot \dot{y}_2 \tag{5.187}$$

where:

- Eq. (5.185) is an additional constraint.
- Eq. (5.186) introduces two new variables: $\ddot{x}$ and $\dot{y}$.
- Eq. (5.187) introduces a new variable: $\dot{y}_2$.

Differentiating Eq. (5.185), a new variable appears, $\ddot{x}$. The initial value of this new variable can be selected so that the equation is satisfied for the given initial values of x, $\dot{x}$, y_1 and y_2.

Once the initial values of x, $\dot{x}$, $\ddot{x}$, y_1 and y_2 have been determined, the initial values of $\dot{y}_1$ and $\dot{y}_2$ can be selected so that Eqs. (5.186) and (5.187) are satisfied, respectively.

Differentiating the equations again, the new variables $\dddot{x}$, $\ddot{y}_1$ and $\ddot{y}_2$ appear. Therefore, new constraints on the original variables are not introduced.

Adding Eq. (5.185) to the system and assuming that x is an algebraic variable, the computational causality of the extended DAE system is:

$$
\begin{aligned}
0 &= f_1\left(\left[x\right], u_1, u_2\right) && (5.188)\\
0 &= \frac{\partial f_1}{\partial x} \cdot [derx] + \frac{\partial f_1}{\partial u_1} \cdot \dot{u}_1 + \frac{\partial f_1}{\partial u_2} \cdot \dot{u}_2 && (5.189)\\
0 &= f_2\left(x, derx, [y_1]\right) && (5.190)\\
0 &= f_3\left(x, [y_2]\right) && (5.191)
\end{aligned}
$$

Observe that x and its derivative ($derx$) are algebraic variables, which are calculated from the system equations. The system has zero DoF.

Now, let's analyze the **inverse problem**. The y_1 and y_2 variables are known, and the objective is to calculate u_1 and u_2. However, observe that u_1 and u_2 only appear in the first equation of the system:

$$0 = f_1\left(x, u_1, u_2\right) \tag{5.192}$$

This indicates that the system does not contain enough information to calculate both u_1 and u_2. The DAE system is ill-posed for the inverse problem. It is not possible to calculate the input trajectories that allow to obtain the desired outputs.

Let's search for additional constraints on the initial values of (x, $\dot{x}$, u_1, u_2). The Pantelides algorithm analyzes which variables intervene in the successive derivatives of the system equations, with the purpose of identifying if any of these derivatives is an additional constraint.

Observe that the first equation of the system,

$$0 = f_1\left(x, u_1, u_2\right) \tag{5.193}$$

depends on x, u_1 and u_2. If this equation is differentiated, the obtained equation

$$0 = \frac{\partial f_1}{\partial x} \cdot \dot{x} + \frac{\partial f_1}{\partial u_1} \cdot \dot{u}_1 + \frac{\partial f_1}{\partial u_2} \cdot \dot{u}_2 \tag{5.194}$$

depends, in general, on x, u_1, u_2, $\dot{x}$, $\dot{u}_1$ and $\dot{u}_2$. Two new unknown variables are introduced, which don't appear in any other equation: $\dot{u}_1$ and $\dot{u}_2$. Each of the successive derivatives of this equation introduces two new variables that only appear in it: the successive derivatives of u_1 and u_2. Therefore, differentiating the first equation does not introduce an additional constraint.

Let's analyze the second and third equations of the DAE system:

$$0 = f_2(x, \dot{x}, y_1) \tag{5.195}$$
$$0 = f_3(x, y_2) \tag{5.196}$$

The second equation depends on x, $\dot{x}$. Differentiating, the obtained equation depends in general on x, $\dot{x}$ and $\ddot{x}$. Differentiating again, the obtained equation depends on x, $\dot{x}$, $\ddot{x}$ and $\dddot{x}$. And so on.

On the other hand, the third equation depends on x. Differentiating, the obtained equation depends on x and $\dot{x}$. Differentiating again, the obtained equation depends on x, $\dot{x}$ and $\ddot{x}$. And so on.

Observe that the second equation, its derivative, the third equation and its derivative form a system of four equations with three unknown variables: x, $\dot{x}$ and $\ddot{x}$. If the second and third equations of the DAE are independents, then there does not exist a solution. From the point of view of the automatic control theory, this system is considered uncontrollable: it is not possible to calculate the inputs (u_1 and u_2) that make the system to produce the desired outputs (y_1 and y_2).

As the successive derivatives of the second and third equations impose additional constraints, the Pantelides algorithm does not converge for the inverse problem. The algorithm differentiates these two equations indefinitely.

5.5 Selection of the state variables

In general, the selection of the state variables is not unique. Different sets of variables can be selected as state variables. This is equivalent to say that it is possible to write the system in different ways, so that different sets of variables appear differentiated.

An adequate selection of the state variables can, in some models, improve the precision and reduce the computational load of the simulation. The state variables can be selected by the model developer and the modeling environment. In the latter case, the modeling environment can modify during the simulation the selection of the state variables. This feature is known as **dynamic selection of the states**.

This section is structured into three parts. A symbolic manipulation technique for selecting the state variables is described in Section 5.5.1. Dynamic selection of the states is addressed in Section 5.5.2. Finally, the state variable selection by the model developer in Modelica is discussed in Section 5.5.3.

5.5.1 Manipulation of the DAE system

Let's consider the following DAE system

$$\mathbf{F}\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, t\right) = 0 \tag{5.197}$$

where $\mathbf{x}$ represents the variables that appear differentiated and $\mathbf{y}$ the variables that don't. Suppose that we want to select as state variables the variables $(\mathbf{x}_1, \mathbf{y}_1)$. The technique described below allows to manipulate the system so that only $(\mathbf{x}_1, \mathbf{y}_1)$ appear differentiated. The resultant system is:

$$\mathbf{G}\left(\mathbf{x_1}, \mathbf{y_1}, \dot{\mathbf{x}}_\mathbf{1}, \dot{\mathbf{y}}_\mathbf{1}, \mathbf{x_2}, \mathbf{y_2}, t\right) = 0 \tag{5.198}$$

where $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$, $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$. The technique consists of the following steps:

1. The following dummy equations are added to the system

$$\dot{\mathbf{y}}_1 = \mathbf{a} \tag{5.199}$$

   where $\mathbf{a}$ is a vector of dummy variables. The $\mathbf{a}$ and $\mathbf{y_1}$ vectors have the same number of components. Observe that these dummy equations don't modify the system solution. As the dummy variables only intervene in the dummy equations, the dummy equations must be employed to calculate the dummy variables.

2. The extended system, composed of the original system and the dummy equations,

$$\mathbf{F}\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, t\right) = 0 \tag{5.200}$$

$$\dot{\mathbf{y}}_1 = \mathbf{a} \tag{5.201}$$

   is a high-index DAE system. Therefore, the next step is to reduce its index, selecting $(\mathbf{x}_1, \mathbf{y}_1)$ as state variables.

The following example illustrates the application of this technique. Consider a control volume that contains a perfectly mixed gas. Establishing an energy balance for the control volume, the obtained model has the following form

$$E = f(T) \tag{5.202}$$
$$\frac{dE}{dt} = g(T,Q) \tag{5.203}$$
$$Q = h(T) \tag{5.204}$$

where E is the internal energy of the gas stored within the control volume, T is the gas temperature, and Q is the heat flow rate exchanged with the environment. The f function, which relates the temperature and internal energy of the stored gas, is in general a non-linear function.

The internal energy (E) of the gas appears differentiated in the model. Selecting E as state variable, the model has the following computational causality:

$$E = f([T]) \tag{5.205}$$
$$\left[\frac{dE}{dt}\right] = g(T,Q) \tag{5.206}$$
$$[Q] = h(T) \tag{5.207}$$

Observe that the gas temperature is calculated from solving the first equation. If f is a non-linear function, this computation can be computationally expensive. Another approach is to select the temperature (T) as state variable, instead of the energy (E). In this way, the first equation is employed for calculating the energy, which is given explicitly.

Let's manipulate the model for obtaining the temperature as the only differentiated variable. A dummy equation, Eq. (5.211), is included in the model.

$$E = f(T) \tag{5.208}$$
$$\frac{dE}{dt} = g(T,Q) \tag{5.209}$$
$$Q = h(T) \tag{5.210}$$
$$\frac{dT}{dt} = a \tag{5.211}$$

As the dummy variable a only intervenes in the dummy equation, Eq. (5.211) must be employed for calculating a. Including the dummy equation does not modify the value of the E, T and Q variables.

The extended DAE system is structurally singular: two variables appear differentiated (E, T) and the system has only one DoF. Let's replace the derivatives by dummy variables:

$$\frac{dE}{dt} \quad \rightarrow \quad derE \qquad\qquad \frac{dT}{dt} \quad \rightarrow \quad derT \tag{5.212}$$

and reduce the DAE system index. To this end, the derivative of Eq. (5.208) is added to the system. It is obtained:

$$E = f(T) \tag{5.213}$$
$$derE = g(T,Q) \tag{5.214}$$
$$Q = h(T) \tag{5.215}$$
$$derT = a \tag{5.216}$$
$$derE = \frac{df(T)}{dT} \cdot derT \tag{5.217}$$

Selecting the temperature as state variable, assigning the computational causality and sorting the equations, it is obtained:

$$T \quad \text{state variable} \tag{5.218}$$
$$[E] = f(T) \tag{5.219}$$
$$[Q] = h(T) \tag{5.220}$$
$$[derE] = g(T,Q) \tag{5.221}$$
$$derE = \frac{df(T)}{dT} \cdot [derT] \tag{5.222}$$
$$derT = [a] \tag{5.223}$$

The dummy equation has already served its purpose and can be removed from the model. The obtained model is composed of Eqs. (5.219) – (5.222), with T selected as state variable.

5.5.2 Dynamic selection of state variables

The example discussed in this section illustrates the necessity of using different sets of variables as state variables in different parts of the simulated trajectory. This implies modifying the selection of state variables during the simulation run.

An ideal pendulum oscillating in a vertical plane is depicted in Figure 5.5. The model is described employing Cartesian coordinates x-y. Two forces are exerted on the mass: gravity force $(m \cdot g)$ and tension force (F). The pendulum length is L. The string has negligible mass. The model is shown below. The equations have been labeled as (a), (b), etc. to facilitate referencing them.

$$
\begin{aligned}
& m \cdot \dot{v}_x = -\tfrac{x}{L} \cdot F && \text{(a)}\\
& m \cdot \dot{v}_y = -\tfrac{y}{L} \cdot F - m \cdot g && \text{(b)}\\
& x^2 + y^2 = L^2 && \text{(c)}\\
& \dot{x} = v_x && \text{(d)}\\
& \dot{y} = v_y && \text{(e)}
\end{aligned}
$$

Four variables $(x,\ y,\ v_x,\ v_y)$ appear differentiated in this model, which has two DoF and index-3. In order to reduce the index, Eq. (c) is differentiated twice, and Eqs. (d) and (e) are differentiated once. Adding these equations to the model and simplifying, it is obtained:

$$
\begin{aligned}
& m \cdot \dot{v}_x = -\tfrac{x}{L} \cdot F && \text{(a)}\\
& m \cdot \dot{v}_y = -\tfrac{y}{L} \cdot F - m \cdot g && \text{(b)}\\
& x^2 + y^2 = L^2 && \text{(c)}\\
& \dot{x} = v_x && \text{(d)}\\
& \dot{y} = v_y && \text{(e)}\\
& x \cdot \dot{x} + y \cdot \dot{y} = 0 && \text{(c')}\\
& x \cdot \dot{v}_x + \dot{x}^2 + y \cdot \dot{v}_y + \dot{y}^2 = 0 && \text{(c'',d',e')}
\end{aligned}
$$

Let's consider the four alternative selections of state variables:

$$\{x, v_x\} \qquad \{x, v_y\} \qquad \{y, v_x\} \qquad \{y, v_y\}$$

The computational causality depends on the state variable selection. In the four cases, the $\{F,\ dervx,\ dervy\}$ variables are evaluated from Eqs. (a), (b) and (c",d'e'). As this is common to the four cases, it is omitted in the following discussion. The computational causality of the equations employed to evaluate $\{y,\ v_y,\ derx,\ dery\}$ is as follows:

1. State variables: $\{x, v_x\}$

$$\begin{aligned}
&[derx] = v_x && \text{(d)}\\
&x^2 + [y]^2 = L^2 && \text{(c)}\\
&x \cdot derx + y \cdot [dery] = 0 && \text{(c')}\\
&dery = [v_y] && \text{(e)}
\end{aligned}$$

   If y equals zero, the simulation crashes with a divide-by-zero error when evaluating Eq. (c').

2. State variables: $\{y, v_x\}$

$$\begin{aligned}
&[derx] = v_x && \text{(d)}\\
&[x]^2 + y^2 = L^2 && \text{(c)}\\
&x \cdot derx + y \cdot [dery] = 0 && \text{(c')}\\
&dery = [v_y] && \text{(e)}
\end{aligned}$$

   If y equals zero, a divide-by-zero error is produced with this selection of the state variables when evaluating Eq. (c').

3. State variables: $\{x, v_y\}$

$$\begin{aligned}
&[dery] = v_y && \text{(e)}\\
&x^2 + [y]^2 = L^2 && \text{(c)}\\
&x \cdot [derx] + y \cdot dery = 0 && \text{(c')}\\
&derx = [v_x] && \text{(d)}
\end{aligned}$$

   If x equals zero, the simulation crashes with a divide-by-zero error when Eq. (c') is evaluated.

4. State variables: $\{y, v_y\}$

$$\begin{aligned}
&[dery] = v_y && \text{(e)}\\
&[x]^2 + y^2 = L^2 && \text{(c)}\\
&x \cdot [derx] + y \cdot dery = 0 && \text{(c')}\\
&derx = [v_x] && \text{(d)}
\end{aligned}$$

   If x equals zero, divide-by-zero error is produced when Eq. (c') is evaluated.

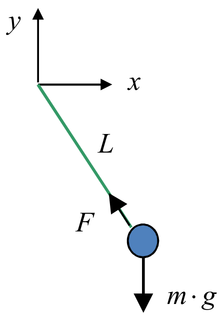
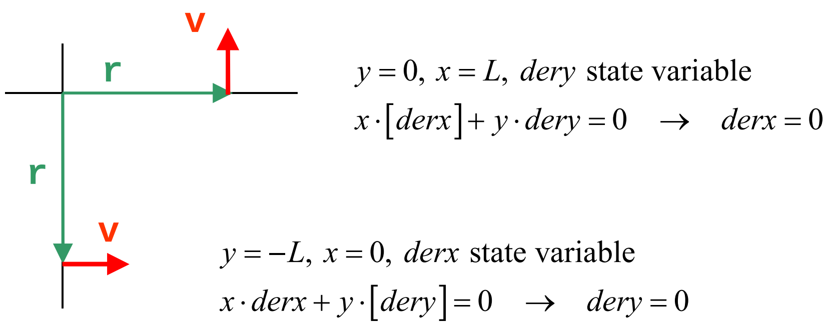


Figure 5.5: Ideal pendulum oscillating in a vertical plane.



v
r
r
v
y = 0, x = L, dery state variable
x·[derx] + y·dery = 0 → derx = 0
y = −L, x = 0, derx state variable
x·derx + y·[dery] = 0 → dery = 0

Figure 5.6: Mandatory causality of pendulum at vertical and horizontal positions.

Therefore, the four selections of the state variables are able to produce a divide-by-zero error when evaluating Eq. (c'). The physical interpretation of Eq. (c') is that the position and velocity vectors are perpendicular. The computational causality that Eq. (c') should have, when the pendulum is in the vertical or horizontal position, is shown in Figure 5.6.

$$x \cdot derx + y \cdot dery = 0 \qquad \leftrightarrow \qquad \mathbf{r} \cdot \mathbf{v} = 0 \qquad \leftrightarrow \qquad \mathbf{r} \perp \mathbf{v}$$

1. When the pendulum is in the vertical position, the perpendicularity condition imposes that the vertical component of the velocity is zero, and does not impose any restriction on the horizontal component of the velocity. For this reason, when the pendulum is in the vertical position, trying to calculate the horizontal component of the velocity from Eq. (c') does not make sense.

2. When the pendulum is in the horizontal position, the perpendicularity condition imposes that the horizontal component of the velocity is zero, and does not impose any restriction on the vertical component of the velocity. Therefore, attempting to calculate the vertical component of the velocity from Eq. (c'), when the pendulum is in the horizontal position, does not make sense.

It is obvious that this difficulty does not appear when the pendulum is modeled using polar coordinates. However, this example illustrates the need of selecting the state variables dynamically during the simulation run.

Dymola supports the dynamic selection of the state variables. To illustrate this feature, let's translate and simulate the model of the pendulum. The model is shown in Modelica Code 5.2. Observe that the initial values of x and vy are specified when the variables are declared:

```
Real x (unit="m",   start=0.9, fixed=true);
Real vy(unit="m/s", start=0,   fixed=true);
```

The initial values are the following: $x_0 = 0.9$, $vy_0 = 0$. The initial value of y is calculated from solving Eq. (c) numerically. The equation admits two solutions:

$$y_0 = \pm\sqrt{L^2 - x_0^2} = \pm\sqrt{1^2 - 0.9^2} = \pm 0.43589 \tag{5.224}$$

Depending on the initial value given to the iterative method employed for solving Eq. (c), the method converges to the positive or negative solution. The initial value for the iterative method is specified in the declaration of the y variable:

```
Real y(unit="m", start=0.5, fixed=false);
```

Setting *fixed* to *false* indicates that the value assigned to the *start* attribute is the initial value of the iterative method employed for calculating the variable at the initial time. Using 0.5 as initial value for the iterative method, it converges to the solution with positive sign. This is, $y_0 = 0.43589$.

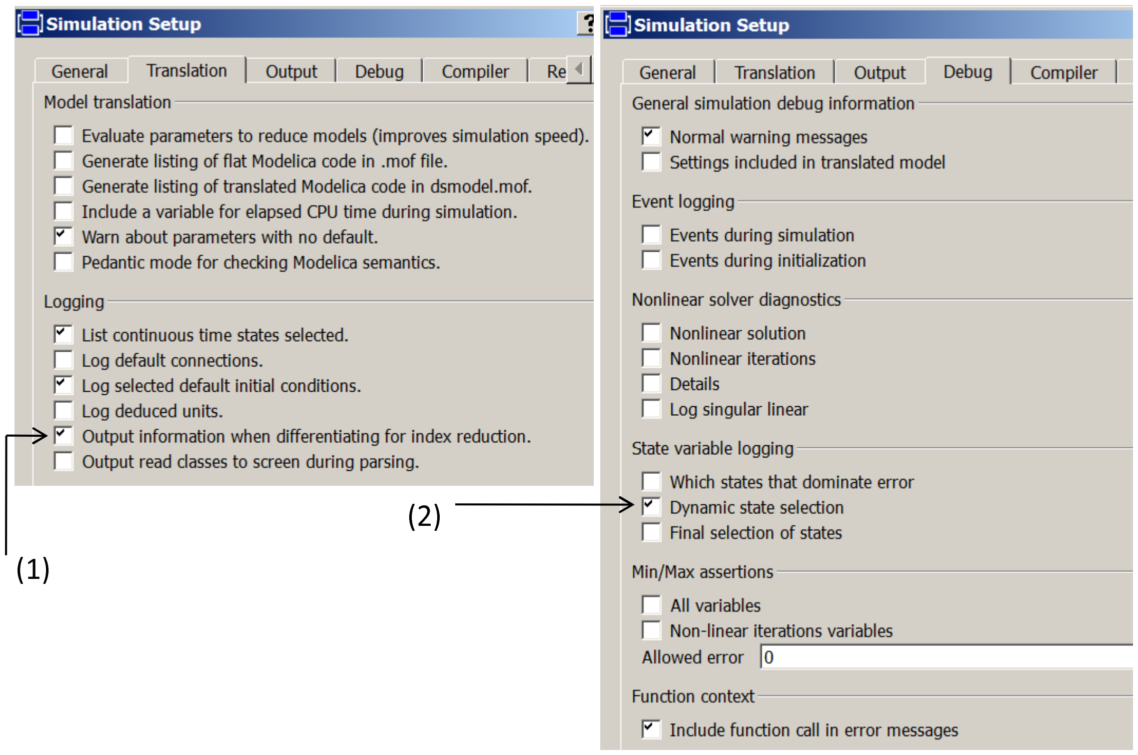
It is shown in Figure 5.7 how to configure the options in the *Simulation Setup* window so that Dymola displays: (1) the equations differentiated and added to the model when the DAE index is reduced; and (2) the state variable selection during the simulation run.

During the translation of the model shown in Modelica Code 5.2, Dymola writes the following text in the log window:

```
Differentiated the equation
x^2+y^2 = L^2;
giving
2.0*(x*der(x)+y*der(y)) = 0.0;

Differentiated the equation
2.0*(x*vx+y*vy) = 0.0;
giving
2.0*(der(x)*vx+x*der(vx)+der(y)*vy+y*der(vy)) = 0.0;
```

```
model pendulumSelecDinVE
  constant Real g(unit="m/s2") = 9.81 ;
  parameter Real L(unit="m") = 1 ;
  parameter Real m(unit="kg") = 5 ;
  Real x(unit="m", start=0.9, fixed=true) ;
  Real y(unit="m", start=0.5, fixed=false) ;
  Real vx(unit="m/s") ;
  Real vy(unit="m/s", start=0, fixed=true) ;
  Real F(unit="N") ;
equation
  m*der(vx) = -x/L*F;
  m*der(vy) = -y/L*F - m*g;
  x^2 + y^2 = L^2;
  der(x) = vx;
  der(y) = vy;
end pendulumSelecDinVE;
```

Modelica Code 5.2: Simple pendulum using Cartesian coordinates.



Simulation Setup
General | Translation | Output | Debug | Compiler | Re

Model translation
- [ ] Evaluate parameters to reduce models (improves simulation speed).
- [ ] Generate listing of flat Modelica code in .mof file.
- [ ] Generate listing of translated Modelica code in dsmodel.mof.
- [ ] Include a variable for elapsed CPU time during simulation.
- [✓] Warn about parameters with no default.
- [ ] Pedantic mode for checking Modelica semantics.

Logging
- [✓] List continuous time states selected.
- [ ] Log default connections.
- [✓] Log selected default initial conditions.
- [ ] Log deduced units.
- [✓] Output information when differentiating for index reduction.
- [ ] Output read classes to screen during parsing.

Simulation Setup
General | Translation | Output | Debug | Compiler

General simulation debug information
- [✓] Normal warning messages
- [ ] Settings included in translated model

Event logging
- [ ] Events during simulation
- [ ] Events during initialization

Nonlinear solver diagnostics
- [ ] Nonlinear solution
- [ ] Nonlinear iterations
- [ ] Details
- [ ] Log singular linear

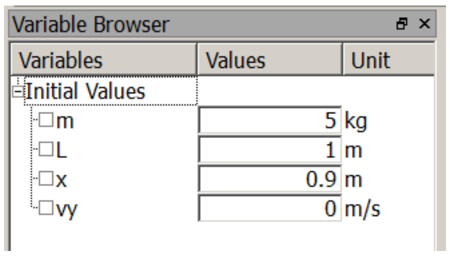
State variable logging
- [ ] Which states that dominate error
- [✓] Dynamic state selection
- [ ] Final selection of states

Min/Max assertions
- [ ] All variables
- [ ] Non-linear iterations variables

Allowed error: 0

Function context
- [✓] Include function call in error messages

Figure 5.7: *Simulation Setup* window of Dymola, where the user configures the information to be written in the log window during the translation and simulation. (1): equations differentiated and added to the model for reducing the DAE index; and (2): state variable selection made during the simulation run.



Variable Browser

Variables	Values	Unit
Initial Values		
m	5	kg
L	1	m
x	0.9	m
vy	0	m/s

Figure 5.8: *Variable Browser* window after translating the model shown in Modelica Code 5.2.

In addition, Dymola writes to the log window the message shown below, indicating that one state variable will be selected from $\{x, y\}$, and one from $\{vx, vy\}$.

```
Selected continuous time states
   Dynamically selected continuous time states
   There are 2 sets of dynamic state selection.
   From set 1 there is 1 state to be selected from:
   x
   y

   From set 2 there is 1 state to be selected from:
   vx
   vy
```

Once the model translation is completed, the variables whose value can be modified are displayed in the *Variable Browser* window (see Figure 5.8).

The model is simulated during during 3 s. The following report about the state variable selection is written to the Dymola log window.

```
Selected at 0:
  y.stateSelect=StateSelect.always
Selected at 0:
  vy.stateSelect=StateSelect.always

Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dynasim))

Selected at 0.534:
  x.stateSelect=StateSelect.always
Selected at 0.534:
  vx.stateSelect=StateSelect.always

Selected at 0.84:
  y.stateSelect=StateSelect.always
Selected at 0.84:
  vy.stateSelect=StateSelect.always

Selected at 1.872:
  x.stateSelect=StateSelect.always
Selected at 1.872:
  vx.stateSelect=StateSelect.always

Selected at 2.184:
  y.stateSelect=StateSelect.always
Selected at 2.184:
  vy.stateSelect=StateSelect.always

Integration terminated successfully at T = 3
```

From time equals zero to 0.534 s, $\{y, vy\}$ have been selected as state variables; from time 0.534 s to 0.84 s, $\{x, vx\}$ have been selected as state variables; etc. Dymola alternates during the simulation run between two state variable selections: $\{y, vy\}$ and $\{x, vx\}$. Dymola employs the first selection when the pendulum trajectory is near the horizontal axis, and the second selection when the trajectory is near the vertical axis.

Dymola names $stateSelect.set1.x[1]$ to the state variable that selects from set1 $= \{x, y\}$, and $stateSelect.set2.x[1]$ to the state variable that selects from set2 $= \{vx, vy\}$. The evolution of $stateSelect.set1.x[1]$ and $stateSelect.set2.x[1]$ is shown in Figure 5.9. It can be observed that $stateSelect.set1.x[1]$ is equal to x in part of the trajectory, and equal to y in the rest of the trajectory; and that $stateSelect.set2.x[1]$ is equal to vx in part of the trajectory and equal to vy in the rest of the trajectory.

5.5.3 Selection by the model developer

There exist several reasons why a model developer may be interested in selecting the model state variables. Some of these reasons may be:

– Improve the accuracy of the model numerical solution.

– Avoid numerical function inversion and, in consequence, increase performance.

– Reduce the number of equations that are nonlinear with respect to the variables to evaluate from them.

– Describe events, given that only the state variables can be reinitialized at event instants using the *reinit* operator (this will be explained in later lessons).

– Avoid dynamic state selection.

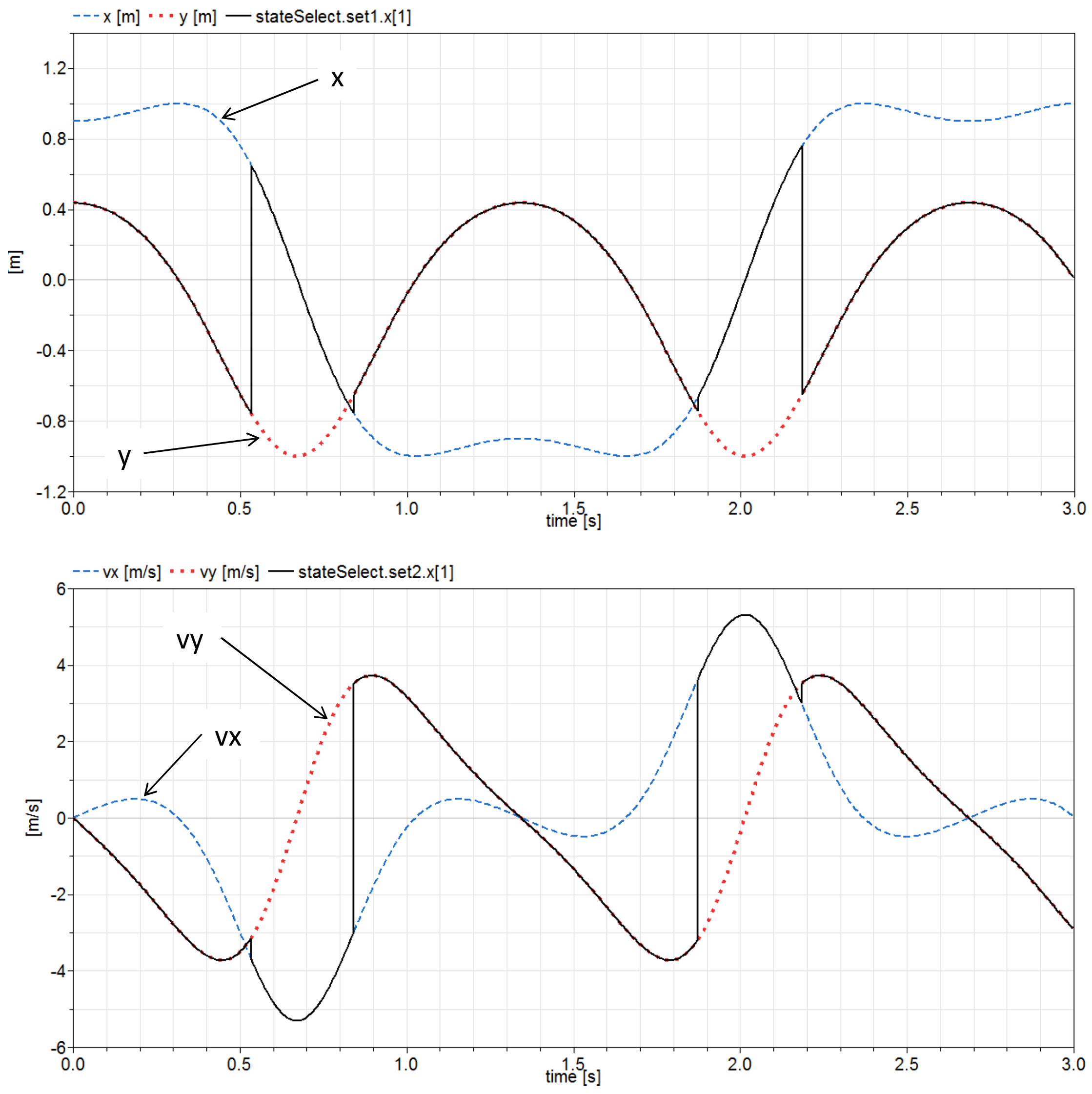


Figure 5.9: Simulation of the pendulum model using Cartesian coordinates, with dynamic selection of the state variables. Dymola defines two sets of variables: set1 = $\{x, y\}$, set2 = $\{vx, vy\}$. The model has two state variables, which Dymola name $\{stateSelect.set1.x[1]$, $stateSelect.set2.x[1]\}$. The $stateSelect.set1.x[1]$ variable is selected among the variables of set1 = $\{x, y\}$, and $stateSelect.set2.x[1]$ among the variables of set2 = $\{vx, vy\}$. During part of the trajectory, $\{stateSelect.set1.x[1], stateSelect.set2.x[1]\}$ is equal to $\{x, vx\}$, and it is equal to $\{y, vy\}$ during the rest of the trajectory. The selection of the state variables is automatically performed by Dymola during the simulation run.

Table 5.1: Possible values of the *stateSelect* attribute of the *Real* variables.

Value	**Meaning**
`StateSelect.never`	Don't select it as state variable.
`StateSelect.avoid`	Avoid it as state variable in favor of the variables having the `default` value.
`StateSelect.default`	If the variable does not appear differentiated in the model, then don't select it as state variable.
`StateSelect.prefer`	Prefer it as state over those having the `default` value.
`StateSelect.always`	Select it as state variable.

The **stateSelect** attribute of the Real variables facilitates the model developer to select the state variables. This is referred to as *static selection* of the state variables, because the selection cannot be changed during the simulation run. The possible values of the attribute are shown in Table 5.1. For instance:

```
Real w ( stateSelect = StateSelect.prefer );
```

A key point is that **the state variable selection in Modelica does not depend on how the model initial conditions are specified**. The value assigned to the *fixed* attribute does not affect the state variable selection. For instance, the initialization of the pendulum model has been performed by setting the value of the $\{x, vy\}$ variables (see Modelica Code 5.2 and Figure 5.8), and the initial selection of the state variables made by Dymola was $\{y, vy\}$.

5.6 Further reading

The definition of DAE index and examples of its calculation can be found in (Brenan et al. 1996).

The Pantelides algorithm is proposed in (Pantelides 1988), and the technique for solving high-index DAE using dummy derivatives in (Mattsson & Söderlind 1992) and (Mattsson & Söderlind 1993).

The structural analysis of DAE systems and the DAE index reduction are explained in Chapter 7 of (Cellier & Kofman 2006).

The selection of state variables in Modelica is discussed in (Mattsson et al. 2000) and (Otter & Olsson 2002), and their dynamic selection by the modeling environment in (Mattsson et al. 2000).

Lesson 6

Numerical methods

6.1 Introduction
6.2 Systems of simultaneous equations
6.3 Numerical solution of ODE
6.4 Numerical solution of DAE
6.5 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Relate modeling hypotheses to algebraic loops, and stiffness.

– Discuss difficulties associated to the symbolic manipulation and the numerical solution of non-linear algebraic loops.

– Perform tearing of small-dimension algebraic loops.

– Classify ODE integration methods attending to the following criteria: explicit and implicit; single-step and multi-step; order; and fixed and variable step size.

– Discuss the principles of the DASSL integration algorithm, and the inline and mixed-mode integration methods.

6.1 Introduction

Once the model has been transformed into an efficiently solvable form, numerical methods are automatically applied to simulate its evolution over time. Some basic concepts related to these numerical methods are introduced in this lesson. The discussion is structured into three parts: solution of algebraic loops, and numerical integration of ODE and DAE.

A rigorous discussion on numerical methods for DAE systems requires of a solid mathematical background and, therefore, is out of the scope of this introductory book. The approach adopted for selecting which material to include in this lesson has been to introduce only those concepts that we consider useful for model developers, in order to use the Modelica modeling environments, and understand the diagnosis and error messages generated by them.

6.2 Systems of simultaneous equations

A set of equations that form a main diagonal block of size greater than 1×1 in the BLT incidence matrix is known as a **system of simultaneous equations**, or equivalently, an **algebraic loop**. Algebraic loops may be linear and non-linear. In the first case, the unknown variables (i.e., the variables that are evaluated from the algebraic loop) intervene linearly in the algebraic loop. In the second case, at least one of the unknown variables intervene non-linearly.

Algebraic loops typically arise of making the simplification that a fast dynamic phenomenon occurs instantaneously. Let's consider, for instance, a system composed of pipes, liquid storage tanks and valves, which is controlled using electronic circuits. This system has hydraulic, mechanical and electronic parts, whose time constants are very different. The response time of the electronic circuits is typically in the order of microseconds, while the response times of the mechanical and hydraulic parts are in the order of 0.1 and 10 seconds, respectively. The time constants of the mechanical and hydraulic parts differ in two orders of magnitude. However, these parts are approximately six orders of magnitude slower than the electronic part.

Systems involving phenomena whose response times differ in three or more orders of magnitude are known as **stiff systems**. If the response times differ in six or more orders of magnitude, then the system is said to be **strongly stiff**.

The slowest time-constant of the system typically determines the time scale of the system's global response, while its fastest time-constant affects the maximum time step length of the numerical integration method. If an explicit integration method is applied for simulating the hydro-electro-mechanical system described previously, the time step should be below one microsecond, while the total simulated time should be in the order of minutes. The number of integration steps is in the order of one hundred million. For this reason, **explicit integration methods** are not efficient for simulating stiff models.

Implicit integration methods are computationally expensive, because they require solving a system of simultaneous equations for each time step. However, implicit methods can allow significantly larger time steps than explicit methods. For stiff systems, implicit methods involve less computational work than explicit methods. For this reason, implicit methods are preferred for solving stiff systems.

As numerical integration of stiff systems is computationally expensive, in some cases it is advantageous to make a modeling hyphothesis consisting in neglecting the fastest dynamics. In this way, stiffness is avoided, but at the expense of establishing an algebraic loop. In other cases, the situation is just the opposite: from a computational cost standpoint, it is more advantageous to describe the dynamics than to assume that the dynamics is instantaneous. This is typically the case when, as a result of the presence of discontinuities, it is not possible to provide adequate initial values for iterating the algebraic loop at certain instants of the simulation.

Observe that it is possible to calculate a solution of the scalar algebraic equation

$$0 = f\left(x\right) \tag{6.1}$$

by solving the ODE

$$\varepsilon \cdot \frac{dx}{dt} = f\left(x\right) \tag{6.2}$$

where a small value is given to ε, and the sign that makes the equation to have a stable solution.

Some basic concepts about the numerical solution of algebraic loops, that need to be known by model developers, are explained in this section. The detailed description of symbolic and numerical algorithms for solving algebraic loops is out of the scope of this text.

6.2.1 Symbolic manipulation of algebraic loops

Linear algebraic loops (i.e., those in which all unknown variables intervene linearly) can be solved automatically by symbolic manipulation. However, if the number of unknown variables is large, it is typically more efficient to employ numerical methods for solving the algebraic loop, than to evaluate the expressions obtained from the symbolic manipulation of the algebraic loop.

Non-linear algebraic loops (i.e., those in which at least one unknown variable intervenes non-linearly), and non-linear scalar equations, are solved using numerical methods. In some simple cases, it is possible to manipulate the non-linear equation in order to obtain explicitly the unknown variable. For instance, y can be obtained explicitly from the equation:

$$x^2 + y^2 = 1 \tag{6.3}$$

Manipulating the equation, the following two solutions for y are obtained:

$$\begin{cases} y = \sqrt{1 - x^2} \\ y = -\sqrt{1 - x^2} \end{cases} \tag{6.4}$$

The problem in this case is that the symbolic manipulator does not known which of these two expressions to employ for calculating y. When Eq. (6.3) is solved numerically, depending on the root-finding algorithm employed, the value used as the initial guess, and how the mathematical problem is formulated, the algorithm may converge to the positive root, to the negative root, or may not converge if started too far away from a root.

6.2.2 Algebraic loops during simulation initialization

As discussed in Section 5.4, the values of all unknown variables need to be calculated at the simulation initial time. This calculation, named **model initialization**, may require the numerical solution of algebraic loops, employing root-finding algorithms. In this case, it is important to provide initial iteration values (initial guess of the solution) as close as possible to the desired solution, so that the iterative algorithm quickly converges to it.

Modelica allows the model developer to provide initial values for iterating the algebraic loops of the model initialization problem. The **start** and **fixed attributes** can be used for this purpose. If the *fixed* attribute is set to *false*, the value assigned to the *start* attribute is used as initial guess by the root-finding algorithm for solving the initialization problem. As the simulation progresses, the variable value at a time instant is used as initial guess for calculating the solution at the next time instant.

In the Modelica models, only continuous-time variables can be calculated by solving algebraic loops. This implies that only variables of the Real type or a type derived from Real can be calculated from an algebraic loop. Discrete-time variables, such as variables of Integer or Boolean types, can not be calculated from algebraic loops.

The solution of the initialization problem is, in some models, problematic. For this reason, some specific-purpose simulation tools implement numerical techniques dedicated to solve the initialization problem. An example is the SPICE electronic circuit simulator. In electronic circuit simulation, one of the challenges is convergence of the root-finding algorithms employed in calculating the circuit operating point. The reason is that, if the initial value of the iterative method is not selected adequately, there is no guarantee that the algorithm converges. The method implemented in SPICE is described below.

Firstly, SPICE assumes that all independent current and voltage sources of the circuit are at zero, and all active devices are in cut-off mode. The circuit operating point is trivial: the voltage at all nodes is zero. Next, a transient analysis is performed, ramping the sources up to their initial values and maintaining these values for a while, so that the circuit is allowed to approach its steady state. Finally, the obtained voltages are employed as initial values for computing the operating point by Newton iteration.

The underlying idea of the SPICE method for solving the initialization problem is of general application. Suppose that solving the model at an initial state I is problematic, but there is another state, T, in which the model can be trivially calculated. Then, the model is simulated, starting from the trivial state T, ramping up the model inputs to their values at the initial state I. Once these values have been reached, they are kept for a while, allowing the model to evolve towards the corresponding steady state. Finally, the state obtained from this simulation is used as initial guess for iterating the initialization problem, this is, calculating the initial state I.

6.2.3 Tearing of non-linear algebraic loops

Tearing is a technique for solving non-linear algebraic loops in which at least one unknown variable appears linearly in one equation. The objective of this technique is to reduce the number of iterated variables.

Let's consider the following system of simultaneous equations.

$$\mathbf{G}(\mathbf{y}) = 0 \tag{6.5}$$

Tearing this algebraic loop consists in writing it in the following form:

$$\begin{cases} \mathbf{G_1}(\mathbf{y_1}, \mathbf{y}_2) = 0 \\ \mathbf{G}_2(\mathbf{y_1}, \mathbf{y}_2) = 0 \end{cases} \tag{6.6}$$

so that the following two conditions are satisfied:

1. The Jacobian matrix $\frac{\partial \mathbf{G_1}}{\partial \mathbf{y_1}}$ is a lower triangular matrix or a block lower triangular (BLT) matrix, where all diagonal blocks represent linear equations or linear systems of simultaneous equations with respect to the unknown variables that are evaluated from them.

2. The dimension of $\mathbf{y_2}$ is as small as possible.

The $\mathbf{y_2}$ variables are known as **tearing variables** and the $\mathbf{G_2}(\mathbf{y_1}, \mathbf{y_2}) = 0$ equations are known as **residue equations**. The tearing variables intervene non-linearly in the residue equations and, in consequence, are calculated from the residue equations employing root-finding algorithms for non-linear equations.

Given an initial value of the tearing variables ($\mathbf{y_2}$), the $\mathbf{y_1}$ variables can be calculated from $\mathbf{G_1}(\mathbf{y_1}, \mathbf{y_2}) = 0$ employing methods for solving linear equations. Using these calculated values of $\mathbf{y_1}$, a new value of $\mathbf{y_2}$ is computed from $\mathbf{G_2}(\mathbf{y_1}, \mathbf{y_2}) = 0$, and so on.

Modelica modeling environments typically support a restricted form of tearing, in which the Jacobian matrix $\frac{\partial \mathbf{G_1}}{\partial \mathbf{y_1}}$ must be lower triangular. Therefore, assuming that $\mathbf{y_2}$ are known, $\mathbf{y_1}$ can be calculated sequentially from the $\mathbf{G_1}(\mathbf{y_1}, \mathbf{y_2}) = 0$ equations, where each unknown variable appears linearly in the equation employed to calculate it.

In general, an algebraic loop admits several possible selections of residue equations and tearing variables. In this case, the tearing variables should be selected among those variables whose initial guess is closer to the solution. Residue equations should be selected attending to their good numerical properties. The tearing algorithms implemented by the Modelica modeling environments select the tearing variables and residue equations by applying heuristic rules.

6.3 Numerical solution of ODE

A distinctive characteristic of continuous-time models is that the state variables can change continuously over time. In any time interval of finite length, there are infinite time instants. As it is impossible to calculate the value of the model variables at infinite time instants, the simulation of continuous-time models is performed by applying algorithms that compute the model variables only at certain time instants. It is performed a **temporal discretization**.

The numerical solution of ordinary differential equations (ODE) is calculated by applying numerical integration methods. Let's consider the following explicit ODE:

$$\frac{dx}{dt} = f\left(x, t\right) \tag{6.7}$$

The value of the x variable is calculated at predefined instants $t_0, t_1, t_2, \ldots$, where t_0 is the initial time of the simulation (see Figure 6.1). The time-interval between two consecutive instants is the **integration step length**: $\Delta t = t_i - t_{i-1}$ with $i = 1, 2, \ldots$ As x is a continuous-time variable, its evolution is typically represented interpolating between the calculated values.

Let's suppose that the simulation has been computed until the instant t_i. This implies that the values of x at the time instants t_0, ..., t_i are known. These are x_0, ..., x_i. The **numerical integration method** has, in general, the following form:

$$x_{i+1} = F\{f(x_{i+1}), f(x_i), f(x_{i-1}), \ldots, x_i, x_{i-1}, x_{i-2}, \ldots\} \tag{6.8}$$

This equation in differences indicates that the value of x at time $i+1$ (x_{i+1}) is calculated from the variable values at previous time instants (x_i, x_{i-1}, x_{i-2}, ...), from the value of the derivative at time $i+1$ ($f(x_{i+1})$), and from the value of the derivative at previous time instants ($f(x_i)$, $f(x_{i-1})$, $f(x_{i-2})$, ...).

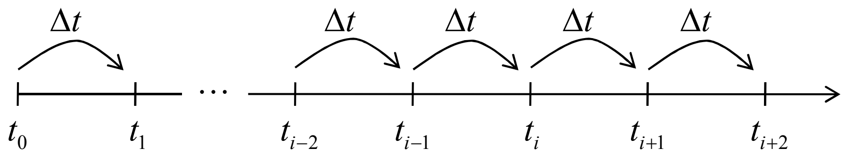


Figure 6.1: Temporal discretization.

The numerical integration methods differ in the form of the F function. They are classified according to the form of F.

– **Explicit and implicit method**. The method is explicit if $f(x_{i+1})$ does not intervene in F. Otherwise, the method is implicit. For instance, the implicit and explicit versions of the **Euler's method** are shown in Table 6.1.

Table 6.1: Euler's methods.

Explicit	$x_{i+1} = x_i + \Delta t \cdot f(x_i)$
Implicit	$x_{i+1} = x_i + \Delta t \cdot f(x_{i+1})$

– **Single-step and multi-step methods**. If values of the variable or its derivative at instants previous to i intervene in F, then it is a multi-step method. If only x_i, $f(x_{i+1})$ and $f(x_i)$ intervene in F, it is a single-step method.

– **Order of the method**. The order of the integration method is the maximum order of the polynomial $x(t)$ that can be exactly represented by x_i. Let's suppose, for instance, that f is a constant value c. The exact solution of the equation

$$\frac{dx}{dt} = c \tag{6.9}$$

is

$$x = x_0 + c \cdot t \tag{6.10}$$

that is a polynomial of degree one in t. An integration method of order one would give exact results. However, an integration method of order one introduces errors if $f = a + b \cdot t$. The exact solution of the equation

$$\frac{dx}{dt} = a + b \cdot t \tag{6.11}$$

is a polynomial of second order

$$x = x_0 + a \cdot t + \frac{1}{2} \cdot b \cdot t^2 \tag{6.12}$$

An integration method of second order would give precise results in this case.

A rigorous description of the ODE integration methods is out of the scope of this textbook. Nevertheless, it is worthwhile to mention that there are basically two ways of obtaining adequate expressions for F.

1. Approximating x with the first few terms of a Taylor series. The number of terms corresponds to the order of the method. An example is the **Runge-Kutta methods**. Some are shown in Table 6.2. Observe that these are explicit, single-step methods.

2. Approximating f with a polynomial, using the values of f calculated in previous instants and, in implicit methods, also the value calculated in the actual instant. Examples of this type of methods are the **Adams-Bashforth methods** and the **Adams-Moulton methods** (see Table 6.3).

The higher the order of the integration method, the more accurate the results and also the faster the simulation, because larger time steps can be used. Integration methods most commonly used in Engineering have order four and five.

The **time step length** Δt is selected searching a compromise between accuracy and computational cost. The smaller the time step Δt, the more accuracy and stability, at the expense of increased computational cost and longer simulation runtimes.

The error associated to a certain value of Δt can be estimated by comparing the results obtained using this value with the results obtained using a smaller time step, for instance, $\frac{\Delta t}{2}$. If the difference is negligible for the purpose of the study, then the value of Δt is adequate. Otherwise, the results obtained using $\frac{\Delta t}{2}$ and $\frac{\Delta t}{4}$ are compared. If the error is negligible, then $\frac{\Delta t}{2}$ is used. If the error is too large. the results obtained using $\frac{\Delta t}{4}$ and $\frac{\Delta t}{8}$ are compared, and so on.

This automatic method of step size adjustment is conceptually simple, but it is not efficient from the computational standpoint. For this reason, the variable step size methods don't employ it.

A more efficient procedure is to employ two embedded integration algorithms, one with higher precision (higher order) than the other. The difference between the

Table 6.2: Some integration methods of Runge-Kutta.

Runge-Kutta 1st order (explicit Euler)	$x_{i+1} = x_i + \Delta t \cdot f(x_i, t_i)$
Runge-Kutta 2nd order	$k_1 = \Delta t \cdot f(x_i, t_i)$ $k_2 = \Delta t \cdot f(x_i + k_1, t_i + \Delta t)$ $x_{i+1} = x_i + \frac{1}{2} \cdot (k_1 + k_2)$
Runge-Kutta 4th order	$k_1 = \Delta t \cdot f(x_i, t_i)$ $k_2 = \Delta t \cdot f\left(x_i + \frac{k_1}{2}, t_i + \frac{\Delta t}{2}\right)$ $k_3 = \Delta t \cdot f\left(x_i + \frac{k_2}{2}, t_i + \frac{\Delta t}{2}\right)$ $k_4 = \Delta t \cdot f(x_i + k_3, t_i + \Delta t)$ $x_{i+1} = x_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$

Table 6.3: Some integration methods based on polynomial approximations. Note that $f(x_i, t_i)$ is abbreviated as f_i.

Order	**Adams-Bashforth**
1	$x_{i+1} = x_i + \Delta t \cdot f_i$
2	$x_{i+1} = x_i + \frac{\Delta t}{2} \cdot (3 \cdot f_i - f_{i-1})$
3	$x_{i+1} = x_i + \frac{\Delta t}{12} \cdot (23 \cdot f_i - 16 \cdot f_{i-1} + 5 \cdot f_{i-2})$
4	$x_{i+1} = x_i + \frac{\Delta t}{24} \cdot (55 \cdot f_i - 59 \cdot f_{i-1} + 37 \cdot f_{i-2} - 9 \cdot f_{i-3})$
Order	**Adams-Moulton**
1	$x_{i+1} = x_i + \Delta t \cdot f_{i+1}$
2	$x_{i+1} = x_i + \frac{\Delta t}{2} \cdot (f_{i+1} + f_i)$
3	$x_{i+1} = x_i + \frac{\Delta t}{12} \cdot (5 \cdot f_{i+1} + 8 \cdot f_i - f_{i-1})$
4	$x_{i+1} = x_i + \frac{\Delta t}{24} \cdot (9 \cdot f_{i+1} + 19 \cdot f_i - 5 \cdot f_{i-1} + f_{i-2})$

Table 6.4: Runge-Kutta-Fehlberg method(4^{th}-5^{th} order).

	$k_1 = \Delta t \cdot f\left(x_i, t_i\right)$
	$k_2 = \Delta t \cdot f\left(x_i + \frac{k_1}{4}, t_i + \frac{\Delta t}{4}\right)$
	$k_3 = \Delta t \cdot f\left(x_i + \frac{3}{32} \cdot k_1 + \frac{9}{32} \cdot k_2, t_i + \frac{3}{8} \cdot \Delta t\right)$
	$k_4 = \Delta t \cdot f\left(x_i + \frac{1932}{2197} \cdot k_1 - \frac{7200}{2197} \cdot k_2 + \frac{7296}{2197} \cdot k_3, t_i + \frac{12}{13} \cdot \Delta t\right)$
	$k_5 = \Delta t \cdot f\left(x_i + \frac{439}{216} \cdot k_1 - 8 \cdot k_2 + \frac{3680}{513} \cdot k_3 - \frac{845}{4104} \cdot k_4, t_i + \Delta t\right)$
	$k_6 = \Delta t \cdot f\left(x_i - \frac{8}{27} \cdot k_1 + 2 \cdot k_2 - \frac{3544}{2565} \cdot k_3 + \frac{1859}{4104} \cdot k_4 - \frac{11}{40} \cdot k_5, t_i + \frac{\Delta t}{2}\right)$
4th order	$x_{i+1} = x_i + \frac{25}{216} \cdot k_1 + \frac{1408}{2565} \cdot k_3 + \frac{2197}{4104} \cdot k_4 - \frac{k_5}{5}$
5th order	$x_{i+1} = x_i + \frac{16}{135} \cdot k_1 + \frac{6656}{12825} \cdot k_3 + \frac{28561}{56430} \cdot k_4 - \frac{9}{50} \cdot k_5 + \frac{2}{55} \cdot k_6$

results obtained from these two algorithms can be considered as an estimation of the error made by the algorithm of lower order.

This strategy can be employed for programming a variable step size algorithm: the error is estimated in each integration step, and if the maximum error is exceeded, then the algorithm steps back and reduces the step size. A strategy to increment the step size can also be implemented. For instance, if the error is below a certain limit during a predefined number of consecutive steps, then the step size is doubled.

For instance, the **Runge-Kutta-Fehlberg method** (RKF45) is a fourth-order method embedded within a fifth-order method (see Table 6.4). The error of the fourth-order method is estimated subtracting the results obtained of both methods. The method can be implemented with a variable step size: the algorithm estimates the error, and consequently reduces or increments the step size.

The **BDF methods** (*Backward Differentiation Formula*) have the following equation:

$$x_{i+1} = \sum_{k=0}^{q-1} \alpha_k \cdot x_{i-k} + \beta_0 \cdot f\left(x_{i+1}, t_{i+1}\right) \cdot \Delta t \tag{6.13}$$

where q is the order of the method, and α_k, β_0 are constants, dependent on the order, selected so that the resulting algorithm has good numerical properties. At the beginning of the simulation, only the initial value x_0 is known. Therefore, the algorithm starts with order one. As the solution is being calculated at successive instants, the order may be incremented. As the method also adjusts the time step size, this is a variable order and variable step size integration method.

6.4 Numerical solution of DAE

Some basic concepts of an integration algorithm named **DASSL** are introduced in this section. DASSL is recognized as one of the most robust methods for numerical solution of DAE systems and variants of DASSL are supported by most Modelica modeling environments. In particular, a variant of DASSL is the by-default integration algorithm of Dymola.

In addition, a summary of the ideas behind two numerical integration techniques named **inline integration** and **mixed-mode integration** is provided in this section. These techniques may improve the simulation efficiency, taking advantage of the available knowledge about the computational structure of the model, and the time constants of its state variables.

6.4.1 DASSL

DASSL is a BDF (Backward Difference Formulae) method, with variable order and step size. Newton's iteration method is applied by DASSL for solving the obtained nonlinear system of simultaneous equations.

Let's consider the following DAE system, where $\mathbf{y}(t)$ is the vector of unknown variables,

$$\mathbf{f}(\dot{\mathbf{y}}, \mathbf{y}, t) = 0 \tag{6.14}$$

and the following discretization scheme, which is employed by many implicit integration algorithms (e.g., BDF methods),

$$\mathbf{y} = h \cdot \dot{\mathbf{y}} + \text{old}(\mathbf{y}) \tag{6.15}$$

where the $\mathbf{y}$ and $\dot{\mathbf{y}}$ vectors represent the values at the new instant; and h is a known scalar. The dependence of h with respect to the time step length varies from one algorithm to another. old$(\mathbf{y})$ is a function of the known values of $\mathbf{y}$, calculated at the previous time instants. For instance, the third order BDF discretization falls within this category:

$$\mathbf{y}_{n+1} = \underbrace{\frac{6}{11} \cdot \bar{h}}_{h} \cdot \dot{\mathbf{y}}_{n+1} + \underbrace{\left(\frac{18}{11} \cdot \mathbf{y}_n - \frac{9}{11} \cdot \mathbf{y}_{n-1} + \frac{2}{11} \cdot \mathbf{y}_{n-2}\right)}_{\text{old}(\mathbf{y})} \tag{6.16}$$

where $\bar{h}$ is the time step length.

Applying the discretization (6.15) to the DAE system (6.14), it is obtained:

$$\mathbf{f}\left(\frac{\mathbf{y} - \text{old}\left(\mathbf{y}\right)}{h}, \mathbf{y}, t\right) = 0 \tag{6.17}$$

Newton's method is used in DASSL for solving $\mathbf{y}$ from the nonlinear system of simultaneous equations (6.17). The Newton's method iterates over the index k:

$$\mathbf{y}^{k+1} = \mathbf{y}^{k} - \mathbf{J}^{-1} \cdot \mathbf{f}\left(\frac{\mathbf{y}^{k} - \text{old}\left(\mathbf{y}\right)}{h}, \mathbf{y}^{k}, t\right) \tag{6.18}$$

$\mathbf{J}^{-1}$ represents the matrix inverse of the Jacobian matrix. The Jacobian matrix of $\mathbf{f}$ is defined as:

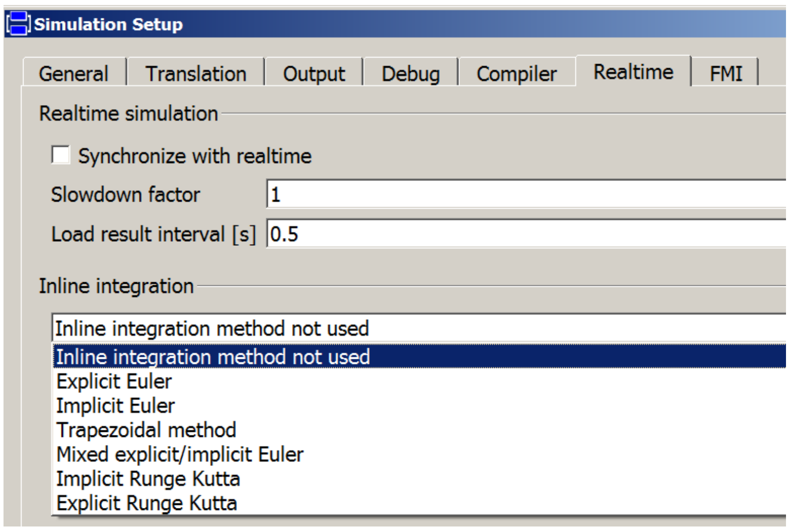
$$\mathbf{J} = \frac{d\mathbf{f}}{d\mathbf{y}} = \frac{\partial \mathbf{f}}{\partial \mathbf{y}} + \frac{1}{h}\frac{\partial \mathbf{f}}{\partial \dot{\mathbf{y}}} \tag{6.19}$$

As evaluating the inverse of the Jacobian matrix is computationally expensive, the inverse matrix calculated at a time instant is employed in as many successive time steps as possible, before being recalculated. This calculation can be performed by applying symbolic manipulation techniques or numerically.

6.4.2 Inline integration

Inline integration is a technique aimed to improve the simulation performance. The step equations of the integration method are explicitly included in the model. As a result, a discretized version of the model is obtained. The computational causality is analyzed, considering as unknown variables the variable values at the new instant. The BLT incidence matrix is obtained and the model is evaluated according to it. The linear equations are manipulated, to obtain explicitly the unknown variables. The tearing technique is applied to the nonlinear algebraic loops. The Newton's method is applied for solving the nonlinear equations. In general, this technique avoids applying the Newton's method to the complete system, as done in Eq. (6.18).

Dymola supports inline integration. The implemented integration methods are shown in Figure 6.2. The method has to be selected before performing the model translation.


Simulation Setup

General | Translation | Output | Debug | Compiler | Realtime | FMI

Realtime simulation

☐ Synchronize with realtime

Slowdown factor: 1

Load result interval [s]: 0.5

Inline integration

Inline integration method not used

- Inline integration method not used
- Explicit Euler
- Implicit Euler
- Trapezoidal method
- Mixed explicit/implicit Euler
- Implicit Runge Kutta
- Explicit Runge Kutta

Figure 6.2: Inline integration methods supported by Dymola version 2017.

6.4.3 Mixed-mode integration

To choose between an explicit or implicit integration method implies to choose between using small integration steps or solving an algebraic loop in each integration step. This motivates the idea of finding a midpoint between implicit and explicit methods. The idea is to divide the system into two parts:

1. A fast system, with the smallest possible dimension, that can be solved using an implicit integration method.

2. A slow system that can be solved using an explicit integration method.

Mixed-mode and inline integration methods may be applied in combination. In this case, an explicit integration method is applied to every state variable, except to those state variables that the model developer has explicitly defined as “fast states”. The implicit integration method would be applied to these fast states.

6.5 Further reading

Tearing of nonlinear systems of simultaneous equations is discussed in (Elmqvist & Otter 1994) and (Cellier & Kofman 2006).

DASSL is described in (Brenan et al. 1996).

Inline integration is discussed in (Elmqvist et al. 1995), and compared with other numerical integration techniques in (Elmqvist et al. 2002). Mixed-mode integration in Modelica is analyzed in (Schiela & Olsson 2000).

Part III

Hybrid system modeling and simulation

Lesson 7

Hybrid system specification

7.1 Introduction
7.2 The OHM formalism
7.3 Model specification and simulation algorithm
7.4 Model specification and Modelica description
7.5 Models with a variable structure
7.6 Model initialization
7.7 Further reading

Learning objectives

After studying the lesson, students should be able to:

- Formulate hybrid-DAE models according to the OHM formalism.
- Given the description of a hybrid-DAE model according to the OHM formalism, formulate the simulation algorithm of the model.
- Describe events in Modelica using when and if clauses.
- Describe variable structure models in Modelica.
- Describe model initialization conditions in Modelica.

7.1 Introduction

The state of hybrid models is described using both continuous-time and discrete-time variables. The hybrid model state evolves over time as a continuous change in the value of the continuous-time state variables, and as instantaneous changes in the total state, continuous and discrete, named events. The simulation algorithm of hybrid models is devised to switch between the solution of the continuous-time problem, and the execution of the events.

A formalism for hybrid models, specially intended to facilitate their simulation, is described in this section. The relationship of the formalism with the simulation algorithm, and with the Modelica description is discussed. The Modelica features for describing instantaneous changes in the model state, and changes in the model structure, are described. Modelica support to models with a variable structure is discussed. Finally, initialization of Modelica models is explained.

7.2 The OHM formalism

There are several formalisms for describing hybrid models. The formalism discussed in this section is based on the OHM (Omola Hybrid Model) formalism, which was proposed in the early 1990s together with the Omola modeling language. Omola is nowadays no longer used, but it had a relevant influence on the first proposal, and initial development of the Modelica language.

A hybrid model M is represented as the following tuple (i.e., finite ordered sequence of elements):

$$M = \langle \mathbf{q}, \mathbf{x}, \mathbf{y}, E, G, H, \Phi, \Delta \rangle \tag{7.1}$$

where the tuple elements are defined as described below.

$\mathbf{q} = \left\{ q_1, ..., q_{n_q} \right\}$ is a vector that contains the model **discrete-time variables**. Discrete-time variables can be of real, integer, Boolean and string type.

$\mathbf{x} = \{ x_1, ..., x_{n_x} \}$ is a vector that contains the **continuous-time state variables**. Continuous-time variables can only be of real type.

$\mathbf{y} = \left\{ y_1, ..., y_{n_y} \right\}$ is a vector that contains the **continuous-time algebraic variables**. As indicated previously, these variables are of real type.

$E = \{e_1, ..., e_{n_e}\}$ is a set that contains all the possible **types of events**.

$G = \left\{g_1, ..., g_{n_g}\right\}$ is the set of expressions that define the **continuous-time behavior** of the model. Each of these expressions represents a continuous-time equation, so that the model equations are:

$$g_i\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{q_t}, t\right) = 0 \qquad \text{with } i = 1, \ldots, n_g \tag{7.2}$$

Continuous-time equations must be satisfied at any time of the simulation, including the model initialization, and the event execution.

The $\mathbf{q_t}$ notation indicates that the value of the discrete-time variables $\mathbf{q}$ is updated at the event instants, being constant between any two consecutive events. From the point of view of the numerical integration algorithm employed for solving the continuous-time part of the model, discrete-time variables have constant known values. The DAE system that describes the continuous-time part of the model is composed of the Eqs. (7.2) and can be represented as:

$$\mathbf{g}\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{q_t}, t\right) = 0 \tag{7.3}$$

Note that we are assuming that the DAE system has been manipulated (if necessary), so that the variables that appear differentiated are the state variables.

$H = \{h_1, ..., h_{n_h}\}$ is a set of Boolean expressions, named **invariant expressions**, that have the following form:

$$h_i\left(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{q_t}, t\right) \tag{7.4}$$

Invariant expressions divide the state space into two regions:

1. The set of admissible states, which are those that make the value of every Boolean expression h_i to be *true*.
2. The set of non-admissible states, which are those that make the value of at least one of the Boolean expression to be *false*.

When the model trajectory crosses the boundary between these two regions, exiting from the set of admissible states, an event is triggered. Therefore, the invariant expressions describe (when the expression value changes from *true* to *false*) trigger conditions of events.

The Boolean complements of the invariant expressions are named **event conditions**. An event condition that depends on at least one continuous-time variable is named **continuous-time event condition**. If an event condition depends only on discrete-time variables, then it is named **discrete-time event condition**. Time events have invariant expressions of the form:

$$h_i \;=\; t < t_i \tag{7.5}$$

where t represents the time variable, and t_i is a discrete-time variable representing the future time in which the event will be triggered.

Φ : $H \rightarrow E$ is a function that **associates an event type to each invariant expression**. When the value of h_i changes from *true* to *false*, the event type associated to h_i is triggered.

$\Delta = \{\boldsymbol{\delta_1}, ..., \boldsymbol{\delta_{n_e}}\}$ is a set of vector expressions that describes the instantaneous change in the model variables produced by the **execution** of each event type. These vector expressions describe vector equations with the following form:

$$\boldsymbol{\delta_i}\left(\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}, \mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}, t_e\right) = 0 \tag{7.6}$$

where $\{\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}\}$ represents the value of the variables after the event execution, $\{\mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}\}$ the value before the event execution, and t_e is the value of the time variable at the event execution instant.

At the event execution time t_e, a discontinuous change in the model variables takes place. The value of the model variables changes from $\{\mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}\}$ to $\{\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}\}$. The former are referred to as the **previous values** and the latter as the **new values**.

The vector expression $\boldsymbol{\delta_i}$ is associated to the event e_i. Being known the value of the variables before the event, the event execution is performed as follows: the vector equation associated to the event, together with all the continuous-time equations of the model, are solved jointly to calculate the value of the variables after the event.

7.3 Model specification and simulation algorithm

The formal specification described in Section 7.2 is closely related to the simulation algorithm implemented by the Modelica modeling environments. This algorithm, which was outlined in Section 1.4, is composed of the following parts.

1. The **solution of the continuous-time problem**, which is described by the DAE system (7.3). As described in Lesson 6, the numerical solution of the DAE system implies solving nonlinear systems of simultaneous equations, and performing numerical integration. The discrete-time variables have constant known values during the solution of the continuous-time problem. By definition, the values of the discrete-time variables only change when executing the events.

2. The **detection of events**, which is carried out by checking the invariant expressions during the solution of the continuous-time problem. When an invariant expression changes from *true* to *false*, the numerical solution of the continuous-time problem is stopped, and an iterative algorithm for finding the trigger time of the event is started.

3. The **determination of the event trigger time**. As the numerical integration of the DAE system advances in time steps, the event can be detected at a time later than its trigger time. Therefore, when an event is detected, an iterative method is employed to locate the event trigger time within the last integration step. A time interval is obtained, satisfying that the interval contains the event trigger time, and the interval length is below a certain tolerance. It is assumed that the event trigger time is the right limit of the interval, and is named t_e.

4. The **execution of the event**. The new values of the model variables, calculated at the event time as a result of executing the event, must be consistent initial values for the continuous-time problem, which will be resumed after executing the event. These new values must satisfy all the equations that describe the continuous-time behavior of the model. For this reason, the event execution is also referred to as **solving the restart problem**. The execution of the e_i event consists in calculating $\{\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}\}$ solving the following system of equations:

$$\begin{aligned} \delta_i\left(\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}, \mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}, t_e\right) &= 0 &\quad (7.7)\\ \mathbf{g}\left(\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}, t_e\right) &= 0 &\quad (7.8)\end{aligned}$$

The restart problem expressed in this form may be difficult to solve, as it contains not only unknown variables of real type, but also of integer and Boolean types. The restart problem is easier to solve expressed as follows:

$$\mathbf{q_a} = \delta_{i,1}\left(\mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}, t_e\right) \tag{7.9}$$

$$\mathbf{x_a} = \delta_{i,2}\left(\mathbf{q_a}, \mathbf{x_b}, \dot{\mathbf{x}}_\mathbf{b}, \mathbf{y_b}, \mathbf{q_b}, t_e\right) \tag{7.10}$$

$$\mathbf{g}\left(\mathbf{x_a}, \dot{\mathbf{x}}_\mathbf{a}, \mathbf{y_a}, \mathbf{q_a}, t_e\right) = 0 \tag{7.11}$$

Being known the previous values of the model variables, the new values of the discrete-time variables are trivially calculated from Eq. (7.9). Next, the new values of the continuous-time state variables are trivially calculated from Eq. (7.10). Finally, the new values of the continuous-time algebraic variables and derivatives are calculated from the DAE system (7.11).

7.4 Model specification and Modelica description

Continuing with the discussion on the specification of hybrid DAE models, the description of events in Modelica is explained below. Events are described in Modelica using when clauses, and if sentences and clauses.

When clauses allow to describe changes in the value of discrete-time variables, and to reinitialize (i.e., change discontinuously) the value of continuous-time state variables. A when clause is composed of a **logical expression** describing the clause trigger condition, and a set of equations, named **instantaneous equations**, in which the new values of the variables appear explicitly indicated. The syntax of the when clause is essentially the following:

$$\begin{array}{l} \textbf{when} \ \text{ logical expression } \ \textbf{then} \\ \quad \text{instantaneous equations} \\ \textbf{end when}; \end{array} \tag{7.12}$$

At the time in which the logical expression changes from *false* to *true*, the changes in the variable values described by the instantaneous equations are made. The instantaneous equations have to be written so that the restart problem is formulated in the form described by Eqs. (7.9) – (7.11).

– The equations describing changes in the value of discrete-time variables have to be written as assignments: the new value assigned to the left-hand side variable is calculated evaluating the expression on the right-hand side.

– The value of continuous-time state variables is reinitialized in Modelica using the **reinit** function. This function has two arguments: the state variable to be reinitialized, and the expression employed for calculating the new value.

If sentences allow to describe changes in the model equations. The modeling environment translates automatically the if sentences into equations. For instance, the following if sentence

$$0 = \textbf{if} \ \ \textit{cond} \ \ \textbf{then} \ f1 \ \textbf{else} \ f2; \tag{7.13}$$

indicates that while *cond* equals *true*, it must be satisfied $0 = f1$, and while *cond* equals *false*, it must be satisfied $0 = f2$. This if sentence is automatically translated by the modeling environment into the following equation

$$0 = \alpha \cdot f_1 + (1 - \alpha) \cdot f_2 \tag{7.14}$$

where α is a discrete-time dummy variable, whose value is one while `cond` equals *true*, and zero while `cond` equals *false*. The changes in the value of α can be performed at events. The relationship with the OHM formalism is as follows: Eq. (7.14), which represents the if sentence, belongs to the G set, and the α variable is a component of the $\mathbf{q}$ vector.

With the purpose of making the formal specification of hybrid models more similar to their Modelica description, we will perform the specification as follows:

1. Indicating the discrete-time variables ($\mathbf{q}$), the continuous-time state variables ($\mathbf{x}$), and the continuous-time algebraic variables ($\mathbf{y}$). The dummy variables introduced for translating the if sentences are included in $\mathbf{q}$.
2. Writing the equations (instead of the expressions) that describe the continuous-time behavior, including the equations equivalent to the if sentences.
3. Writing the tuple elements E, H, Φ and Δ in a table with the following characteristics.
   - The table has n_e rows: one row per event type in the E set. The first column of the table contains the consecutive numbering of its rows. The i-th row of the table corresponds to the e_i event type.
   - The second column of the table contains the trigger condition of each event type (instead of the corresponding invariant expression), this is,

the Boolean complements of H. The Φ function associates invariant expression with event types. As the trigger condition of the i-th event type is written in the i-th row of the table, the Φ function is represented in the table.

– The third column of the table contains the instantaneous equations (instead of the vector expression) that describe the change in the values of the discrete-time variables and the continuous-time state variables produced at the event. State variables whose change is not explicitly indicated are assumed to remain constant at the event.

The following examples illustrate how the specification of some hybrid models can be described.

7.4.1 Bouncing ball

Let's see an example of how to describe the formal specification of a hybrid model. The model describes the vertical movement of a ball that falls down due to gravity and bounces on the floor. The gravitational acceleration is assumed to have a constant value, $g = 9.8\ \mathrm{m/s^2}$. The vertical position of the ball (i.e., the vertical distance from the ball center to the floor), and the ball velocity are denoted as h and v respectively.

The references for the ball position and velocity are selected as follows. The floor is assumed to be at rest, and at zero position. Positions above the floor are positive. The velocity of the ball is positive when it moves upward. The following two equations describe the free vertical movement of the ball.

$$\frac{dh}{dt} = v \tag{7.15}$$
$$\frac{dv}{dt} = -g \tag{7.16}$$

The initial position and velocity of the ball are denoted as h_0 y v_0 respectively.

The ball bounces when touches the floor. It is assumed that the direction of the ball velocity is reversed and its magnitude is reduced by 20 %. Naming v_{b} the velocity before the bounce and v_{a} the velocity after the bounce, it is verified when the bounce takes place:

$$v_{\mathrm{a}} = -0.8 \cdot v_{\mathrm{b}} \tag{7.17}$$

For instance, if the ball velocity before bouncing is -10 m/s, then after bouncing is 8 m/s. The negative sign of the velocity indicates that the ball is falling, while the positive sign indicates that it is ascending. Eq. (7.17) can be described in Modelica employing a when clause and the reinit function, as shown below.

$$\begin{aligned}&\textbf{when } h \le 0 \textbf{ then}\\ &\quad \textbf{reinit}(v, -0.8 \cdot v);\\ &\textbf{end when};\end{aligned} \tag{7.18}$$

The **reinit** function has two arguments. The first one is the continuous-time state variable, and the second one is a real type expression. When the reinit function is called, the expression of the second argument is evaluated and the obtained value is assigned to the state variable passed as first argument.

The ball position at the event execution time will be very close to zero, but probably it will not be exactly zero. Including within the when clause another call to the reinit function, the ball position is reinitialized to zero after the event execution.

$$\begin{aligned}&\textbf{when } h \le 0 \textbf{ then}\\ &\quad \textbf{reinit}(v, -0.8 \cdot v);\\ &\quad \textbf{reinit}(h, 0);\\ &\textbf{end when};\end{aligned} \tag{7.19}$$

The Boolean expression $(h \le 0)$ is the logical condition, and the two reinit sentences are the instantaneous equations of the when clause.

The formal specification of this model can be described as follows. The vectors containing the model variables are shown below. As the model does not contain discrete-time variables and continuous-time algebraic variables, the corresponding vectors are empty.

$$\mathbf{q} = \{\ \} \tag{7.20}$$

$$\mathbf{x} = \{h, v\} \tag{7.21}$$

$$\mathbf{y} = \{\ \} \tag{7.22}$$

Table 7.1: Events of the bouncing ball model.

	Event condition	Instantaneous equations
1	$h \leq 0$	$v_a = -0.8 \cdot v_b$ $h_a = 0$

The continuous-time equations are Eqs. (7.15) and (7.16). The events are described in Table 7.1.

This model has only one type of event. Observe that the restart problem for this event consists in calculating $\{h_a, v_a, derh_a, derv_a\}$ from the system of four equations composed by the instantaneous equations of the event (see the second column of Table 7.1), and the two equations that describe the continuous-time behavior, this is, Eqs. (7.15) and (7.16). The system is written below.

$$v_a = -0.8 \cdot v_b \tag{7.23}$$
$$h_a = 0 \tag{7.24}$$
$$derh_a = v_a \tag{7.25}$$
$$derv_a = -g \tag{7.26}$$

7.4.2 Liquid storage tank with drain valve

As a second example, let's consider the system depicted in Figure 7.1. It is composed of a liquid storage tank, an input pipe (connected on the top of the tank), a drain pipe (connected at medium height, h_0), and a valve to regulate the flow through the drain pipe.

The valve opening, represented as V, takes values between 0 (fully closed), and 1 (fully open, allows to pass 100 % of the flow). Let's assume that the input flow of liquid (F_{in}) and the valve opening (V) are known functions of time.

$$F_{in} = f_1(t) \tag{7.27}$$
$$V = f_2(t) \tag{7.28}$$

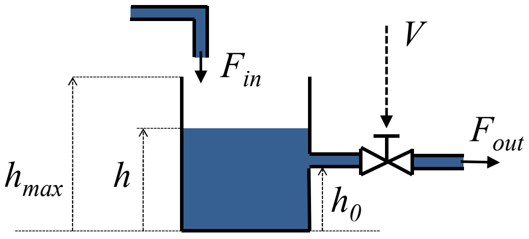


Figure 7.1: Liquid storage tank with drain valve.

The drain pipe is connected at h_0 height. Therefore, liquid flows through the drain pipe only if the following two conditions are simultaneously satisfied: the liquid level in the tank (h) is greater than h_0, and the valve is not closed. The output flow of liquid is described by the following two-branch equation:

$$F_{out} = \begin{cases} 0 & \text{if } h \leq h_0 \\ K \cdot V \cdot \sqrt{h - h_0} & \text{if } h > h_0 \end{cases} \tag{7.29}$$

where K and h_0 are parameters with known values. Eq. (7.29) can be described in Modelica employing an if sentence, which will be automatically translated by the modeling environment into an equation of the form:

$$F_{out} = \alpha \cdot K \cdot V \cdot \sqrt{h - h_0} \tag{7.30}$$

where α is a discrete-time dummy variable, whose value is zero while $h \leq h_0$, and one otherwise (i.e., while $h > h_0$).

Assuming that the liquid has a constant density, the balance of liquid mass in the tank can be expressed as:

$$A \cdot \frac{dh}{dt} = F_{in} - F_{out} \tag{7.31}$$

where the tank section, denoted as A, is a known parameter.

The system has an overflow alarm that is triggered when the liquid level inside the tank becomes greater than a known predefined value, h_{max}. The alarm is described by a Boolean variable named *Alarm*, whose value is calculated evaluating the logical expression $h > h_{max}$.

The formal specification of the model is shown below. The model variables are:

Table 7.2: Events of the tank model.

	Event condition	**Instantaneous equations**
1	$\alpha == 0$ and $h > h_0$	$\alpha_a = 1$
2	$\alpha == 1$ and $h \leq h_0$	$\alpha_a = 0$
3	$Alarm == 0$ and $h > h_{max}$	$Alarm_a = 1$
4	$Alarm == 1$ and $h \leq h_{max}$	$Alarm_a = 0$

$$\begin{aligned} \mathbf{q} &= \{\alpha, Alarm\} && (7.32)\\ \mathbf{x} &= \{h\} && (7.33)\\ \mathbf{y} &= \{F_{in}, F_{out}, V\} && (7.34) \end{aligned}$$

The equations that describe the continuous-time behavior are Eqs. (7.27), (7.28), (7.30) and (7.31). The events are described in Table 7.2. Observe that there are four types of events.

If the change in the value of a state variable is not explicitly indicated by an instantaneous equation, then it is assumed that the value of this state variable does not change at the event. An equation stating that the “before” and “after” values of this state variable are equal is automatically included in the restart problem.

For instance, assuming that the event is triggered at time t_e, the restart problem for the first type of event (see Table 7.2) is:

$$\begin{aligned} \alpha_a &= 1 && (7.35)\\ Alarm_a &= Alarm_b && (7.36)\\ h_a &= h_b && (7.37)\\ F_{in_a} &= f_1(t_e) && (7.38)\\ V_a &= f_2(t_e) && (7.39)\\ F_{out_a} &= \alpha_a \cdot K \cdot V_a \cdot \sqrt{h_a - h_0} && (7.40)\\ A \cdot derh_a &= F_{in_a} - F_{out_a} && (7.41) \end{aligned}$$

Eq. (7.35) is the instantaneous equation indicated in the event table. Eqs. (7.36) and (7.37) indicate that the values of the $Alarm$ and h variables don't change at the event. Eqs. (7.38) – (7.41) describe the continuous-time behavior. The unknown variables to calculate from these seven equations, Eqs. (7.35) – (7.41), are: $\{\alpha_a$, $Alarm_a$, h_a, $derh_a$, F_{in_a}, V_a, $F_{out_a}\}$.

7.4.3 Two-tank and valve system

Let's consider the system depicted in Figure 7.2 that is composed of two liquid storage tanks connected through a valve. The mass of the liquid stored in the tanks is denoted as m_1 and m_2, and the temperature of the liquid as T_1 and T_2. The tank sections, S_1 and S_2, and the gravitational acceleration, g, are known parameters of the model. The mass flow rate from tank 1 to tank 2 is denoted as F^m.

The first tank is modeled writing the mass and energy balances, and the relationship between the pressure at the bottom and the mass of liquid. The heat capacity of the liquid (C_p) is a known parameter of the model.

$$\begin{aligned} \frac{dm_1}{dt} &= -F^m && (7.42) \\ m_1 \cdot C_p \cdot \frac{T_1}{dt} &= -F^m \cdot C_p \cdot (T_f - T_1) && (7.43) \\ p_1 &= \frac{m_1 \cdot g}{S_1} && (7.44) \end{aligned}$$

The second tank is modeled analogously.

$$\begin{aligned} \frac{dm_2}{dt} &= F^m && (7.45) \\ m_2 \cdot C_p \cdot \frac{T_2}{dt} &= F^m \cdot C_p \cdot (T_f - T_2) && (7.46) \\ p_2 &= \frac{m_2 \cdot g}{S_2} && (7.47) \end{aligned}$$

The mass flow rate between the tanks (F^m) is proportional to the valve opening (θ), and to the K_v parameter. We assume that the flow is positive if exits Tank 1 and enters Tank 2. The constitutive relationship of the valve is:

$$F^m = \begin{cases} K_v \cdot \theta \cdot \sqrt{p_1 - p_2} & \text{if} \quad p_1 > p_2 \\ -K_v \cdot \theta \cdot \sqrt{p_2 - p_1} & \text{if} \quad p_1 \leq p_2 \end{cases} \qquad (7.48)$$

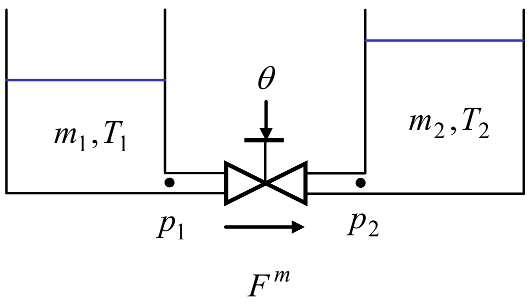


Figure 7.2: Liquid storage tanks connected through a valve.

The temperature of the liquid flowing through the valve (T_f) depends on the flow direction:

$$T_f = \begin{cases} T_1 & \text{if} \quad p_1 > p_2 \\ T_2 & \text{if} \quad p_1 \leq p_2 \end{cases} \tag{7.49}$$

The two-branch equations (7.48) and (7.49) can written as shown below, introducing a dummy variable α whose value is one while $p_1 > p_2$, and zero otherwise.

$$F^m = \alpha \cdot K_v \cdot \theta \cdot \sqrt{p_1 - p_2} + (1 - \alpha) \cdot \left(-K_v \cdot \theta \cdot \sqrt{p_2 - p_1}\right) \tag{7.50}$$

$$T_f = \alpha \cdot T_1 + (1 - \alpha) \cdot T_2 \tag{7.51}$$

The valve opening (θ) is modeled as a discrete-time variable. An initial value is assigned to the valve opening. The valve opening is kept constant during t_0 seconds. Then, the valve opening changes instantaneously to the value θ_0. The t_0 and θ_0 quantities are known parameters of the model. This behavior can be described employing a when clause:

$$\begin{array}{l} \textbf{when } t > t_0 \textbf{ then} \\ \quad \theta = \theta_0; \\ \textbf{end when}; \end{array} \tag{7.52}$$

The formal specification of this model is shown below. The model variables are:

$$\mathbf{q} = \{\theta, \alpha\} \tag{7.53}$$

$$\mathbf{x} = \{m_1, m_2, T_1, T_2\} \tag{7.54}$$

$$\mathbf{y} = \{F^m, T_f, p_1, p_2\} \tag{7.55}$$

Table 7.3: Events of the two-tank and valve model.

	Event condition	**Instantaneous equations**
1	$\alpha == 0$ and $p_1 > p_2$	$\alpha_a = 1$
2	$\alpha == 1$ and $p_1 \leq p_2$	$\alpha_a = 0$
3	$t > t_0$	$\theta_a = \theta_0$

The following equations describe the continuous-time behavior:

$$T_f = \alpha \cdot T_1 + (1-\alpha) \cdot T_2 \tag{7.56}$$
$$F^m = \alpha \cdot K_v \cdot \theta \cdot \sqrt{p_1 - p_2} + (1-\alpha) \cdot \left(-K_v \cdot \theta \cdot \sqrt{p_2 - p_1} \right) \tag{7.57}$$
$$\frac{dm_1}{dt} = -F^m \tag{7.58}$$
$$m_1 \cdot C_p \cdot \frac{dT_1}{dt} = -F^m \cdot C_p \cdot (T_f - T_1) \tag{7.59}$$
$$p_1 = \frac{m_1 \cdot g}{S_1} \tag{7.60}$$
$$\frac{dm_2}{dt} = F^m \tag{7.61}$$
$$m_2 \cdot C_p \cdot \frac{dT_2}{dt} = F^m \cdot C_p \cdot (T_f - T_2) \tag{7.62}$$
$$p_2 = \frac{m_2 \cdot g}{S_2} \tag{7.63}$$

The events are described in Table 7.3. Three types of events have been defined. The two firsts types of event describe the changes in the flow direction. The third type of event describes the abrupt change in the valve opening.

We have assumed that the mass flow through the valve is an algebraic function of the pressure difference. This is equivalent to neglect the inertia of the circulating liquid. Let's now take a different approach in modeling the valve. Instead of neglecting the liquid inertia, we assume now that the linear momentum of the liquid (P) changes over time in response to the pressure difference. The valve model is shown below. When clauses are employed to describe the change in the valve opening, and to reinit the linear momentum when the valve becomes closed. The L parameter represents the effective length of the valve, and K_v^* is a characteristic parameter of the valve.

$$T_f = \begin{cases} T_1 & \text{if } P > 0 \\ T_2 & \text{otherwise} \end{cases} \quad (7.64)$$

$$\frac{dP}{dt} = K_v^* \cdot \theta^2 \cdot (p_1 - p_2) \quad (7.65)$$

$$P = F^m \cdot L \quad (7.66)$$

$$\begin{array}{l} \textbf{when } t > t_0 \textbf{ then} \\ \quad \theta = \theta_0 \\ \textbf{end when}; \end{array} \quad (7.67)$$

$$\begin{array}{l} \textbf{when } \theta \le 0 \textbf{ then} \\ \quad \textbf{reinit}\,(P, 0)\,; \\ \textbf{end when}; \end{array} \quad (7.68)$$

The formal specification of the model is written next. The model variables are:

$$\mathbf{q} = \{\theta, \alpha\} \quad (7.69)$$

$$\mathbf{x} = \{m_1, m_2, T_1, T_2, P\} \quad (7.70)$$

$$\mathbf{y} = \{F^m, T_f, p_1, p_2\} \quad (7.71)$$

where α is a dummy variable introduced to translate into an equation the two-branch equation (7.64), which describes the temperature (T_f) of the liquid flowing through the valve. The continuous-time behavior is described by the following equations.

$$T_f = \alpha \cdot T_1 + (1 - \alpha) \cdot T_2 \quad (7.72)$$

$$\frac{dP}{dt} = K_v^* \cdot \theta^2 \cdot (p_1 - p_2) \quad (7.73)$$

$$P = F^m \cdot L \quad (7.74)$$

$$\frac{dm_1}{dt} = -F^m \quad (7.75)$$

$$m_1 \cdot C_p \cdot \frac{dT_1}{dt} = -F^m \cdot C_p \cdot (T_f - T_1) \quad (7.76)$$

$$p_1 = \frac{m_1 \cdot g}{S_1} \quad (7.77)$$

$$\frac{dm_2}{dt} = F^m \quad (7.78)$$

$$m_2 \cdot C_p \cdot \frac{dT_2}{dt} = F^m \cdot C_p \cdot (T_f - T_2) \quad (7.79)$$

$$p_2 = \frac{m_2 \cdot g}{S_2} \quad (7.80)$$

Table 7.4: Events considering the inertia of the liquid that flows through the valve.

	Event condition	**Instantaneous equations**
1	$\alpha == 0$ and $P > 0$	$\alpha_a = 1$
2	$\alpha == 1$ and $P \leq 0$	$\alpha_a = 0$
3	$t > t_0$	$\theta_a = \theta_0$
4	$\theta \leq 0$	$P_a = 0$

The model events are described in Table 7.4. There have been defined four types of events. The two first types describe the change in the flow direction. The third type describes the abrupt change in the valve opening. The fourth type of event sets to zero the linear momentum of the liquid when the valve becomes closed.

7.5 Models with a variable structure

A model is said to have a **variable structure** if its mathematical description can change during the simulation run. Models of this type can be in different **modes**, being each mode described by a particular system of equations. During the simulation run, transitions between modes are taking place according to predefined conditions, producing the corresponding changes in the model mathematical description. Some examples are discussed next.

7.5.1 Ideal switch

A variable structure model that appears in different domains is the ideal switch. The model has two connectors. Each connector has an across (e) and a through (f) variable. Let's represent the variables of the first connector as e_1/f_1, and the variables of the second connector as e_2/f_2. The through quantity is conserved, but is not accumulated inside the ideal switch: $f_1 = -f_2$. The ideal switch has two modes: *Open* and *Close*. The constitutive relationship of the ideal switch is $f_1 = 0$ while in the *Open* mode, and it is $e_1 = e_2$ while in the *Close* mode.

To illustrate the previous description, a model of an ideal switch in the hydraulic domain is represented in Figure 7.3. The actual mode of the switch is determined by the value of a Boolean variable named *OpenSw*. The connector variables are pressure (p), and volumetric flow rate (F_V).

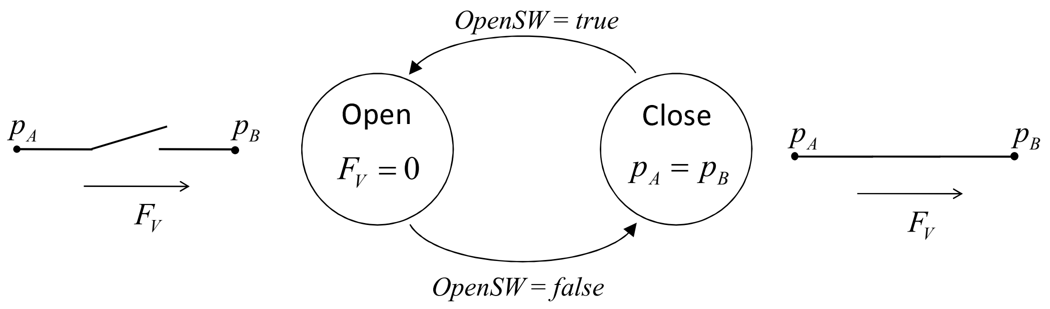


Figure 7.3: Two-mode model of an ideal switch in the hydraulic domain.

– While *OpenSw* equals *true*, the switch is in the **Open** mode: there is no flow through the switch, and the pressure drop is determined by the rest of the hydraulic circuit. The constitutive relationship is:

$$F_V = 0 \tag{7.81}$$

– While *OpenSw* equals *false*, the switch is in the **Close** mode. The pressure drop is zero, and the flow through the switch is determined by the rest of the hydraulic circuit. The constitutive relationship is:

$$p_A = p_B \tag{7.82}$$

As described in the Section 7.4, models with a variable structure can be described in Modelica employing if sentences. These sentences are automatically translated by the modeling environment into equations. The constitutive relationship of the ideal switch represented in Figure 7.3 can be described in Modelica using this if sentence:

$$0 = \textbf{if } \mathit{OpenSw} \textbf{ then } F_V \textbf{ else } p_A - p_B; \tag{7.83}$$

that is automatically translated by the modeling environment into the following equation:

$$0 = \alpha \cdot F_V + (1 - \alpha) \cdot (p_A - p_B) \tag{7.84}$$

where the discrete-event dummy variable α is equal to one while *OpenSw* is *true*, and is equal to zero while *OpenSw* is *false*.

The computational causality of the switch's constitutive relationship depends on the value of the *OpenSw* variable. While the value of *OpenSw* is *true*, the volumetric flow rate is evaluated from the constitutive relationship: $[F_V] = 0$. While *OpenSw* is *false*, the pressure at a connector is evaluated from the constitutive relationship: $[p_A] = p_B$ or $p_A = [p_B]$.

The computational causality of the switch's constitutive relationship can change during the simulation run. Therefore, assigning the computational causality of the complete model, the switch's constitutive relationship will be part of an algebraic loop. The model discussed in Section 7.5.2 illustrates it.

7.5.2 Hydraulic system with relief pipe

Let's consider the model depicted in Figure 7.4. The main hydraulic circuit is composed of a primary pipe connected to a source of liquid. An ideal switch and a secondary pipe are connected in parallel to the primary pipe.

The hydraulic circuit is regulated as follows. During normal operating conditions, the switch is open and all the liquid flows through the primary pipe ($F_{V,2} = 0$). When the volumetric flow rate through the primary circuit is larger than a certain critical value, the switch is closed, circulating part of the liquid flow through the secondary hydraulic circuit ($p_1 = p_2$).

Let's assume for constructing this model that the constitutive relationship of a pipe is Eq. (7.85), where F_V is the volumetric flow rate through the pipe, Δp the pressure drop between the pipe connectors, S is the cross sectional area of the pipe, and c_{desc} is a coefficient whose value can be calculated from Eq. (7.86).

$$\begin{aligned} F_V &= S \cdot c_{desc} \cdot \sqrt{\Delta p} && (7.85) \\ c_{desc} &= \sqrt{\frac{2 \cdot D}{\kappa_{Fanning} \cdot L \cdot \rho}} && (7.86) \end{aligned}$$

The pipe's constitutive relationship can be linearized around an operating point $(F_{V,0}, \Delta p_0)$. It is obtained:

$$F_V = F_{V,0} + \frac{S \cdot c_{desc}}{2 \cdot \sqrt{\Delta p_0}} \cdot (\Delta p - \Delta p_0) \qquad (7.87)$$

This linearized constitutive relationship can be written as:

$$p_A - p_B = R^* + R \cdot F_V \qquad (7.88)$$

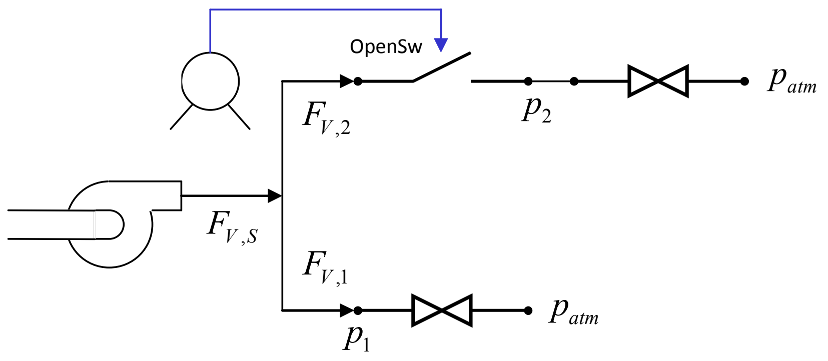


Figure 7.4: Hydraulic system composed of a liquid source, two pipes, and an ideal switch.

The following equations describe the hydraulic circuit shown in Figure 7.4. We have made the simplifying assumption that the atmospheric pressure (p_{atm}) is constant. For the sake of simplicity, the equations to calculate α have not been included in the model.

$$\begin{aligned} p_1 - p_{atm} &= R_1^* + R_1 \cdot F_{V,1} && (7.89)\\ p_2 - p_{atm} &= R_2^* + R_2 \cdot F_{V,2} && (7.90)\\ 0 &= \alpha \cdot F_{V,2} + (1-\alpha) \cdot (p_1 - p_2) && (7.91)\\ F_{V,S} &= F_{V,1} + F_{V,2} && (7.92)\\ F_{V,S} &= f(t) && (7.93) \end{aligned}$$

Assuming that α is an input variable of the model, the computational causality of the model can be calculated. The sorted and solved model is written below.

$$\begin{aligned} [F_{V,S}] &= f(t) && (7.94)\\ [F_{V,2}] &= \frac{(R_1^* - R_2^* + R_1 \cdot F_{V,S})(1-\alpha)}{(R_1 + R_2)(1-\alpha) - \alpha} && (7.95)\\ [F_{V,1}] &= F_{V,S} - F_{V,2} && (7.96)\\ [p_1] &= R_1^* + R_1 \cdot F_{V,1} + p_{atm} && (7.97)\\ [p_2] &= R_2^* + R_2 \cdot F_{V,2} + p_{atm} && (7.98) \end{aligned}$$

The denominator of Eq. (7.95) is different from zero in both modes (i.e., in the cases $\alpha = 0$ and $\alpha = 1$), meaning that the model is valid for the two modes of the ideal switch. If the hydraulic system would contain N ideal switches, the complete model would have the required algebraic loops for it to be valid in any of the 2^N possible combinations of the switches' modes.

In this model, the number of DoF is independent of the switch mode. However, there exist models in which the mode transitions introduce or eliminate constraints on the state variables, which change the DoF number of the model. An example is shown in Section 7.5.3.

7.5.3 Drain pipe

Pipes were modeled in Section 7.5.2 as resistive components, describing the pressure loss caused by the friction between the pipe wall and the liquid. Another phenomenon related to the liquid flow in pipes is the liquid inertia. This phenomenon can be modeled describing that the liquid stores kinetic energy, exhibiting a behavior analogous to the electric inductor.

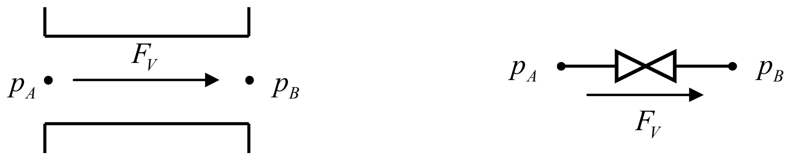
Let's consider a liquid of constant density ρ that flows with uniform velocity v (independent of the spatial coordinates, but time dependent) in a pipe with cross-sectional area S and length L. The linear momentum (P) of the liquid in the pipe can be calculated from Eq. (7.99), where the mass of liquid is equal to $\rho \cdot L \cdot S$, and F_V is the volumetric flow rate ($F_V = S \cdot v$).

$$P = \underbrace{\rho \cdot L \cdot S}_{\text{Mass of liquid}} \cdot v = \rho \cdot L \cdot \underbrace{F_V}_{=S \cdot v} \tag{7.99}$$

The derivative of the liquid's linear momentum is equal to the force exerted on the liquid (Newton's second law of motion). It is assumed that the pressure difference between the pipe terminals is the only force exerted on the liquid that is inside the pipe. The model is represented on the left side of Figure 7.5. In analogy with the electrical domain, the I coefficient calculated in Eq. (7.100) is named inductance.

$$I = \frac{\rho \cdot L}{S} \tag{7.100}$$

On the right side of Figure 7.5, the resistive behavior discussed in Section 7.5.2 is represented. Energy dissipation by friction between the liquid and the pipe wall is described as a nonlinear algebraic relationship involving the pressure drop and the volumetric flow rate.



$$\boxed{\frac{\rho \cdot L}{S} \cdot \frac{dF_V}{dt} = p_A - p_B} \qquad \boxed{p_A - p_B = \left(\frac{1}{S \cdot c_{desc}}\right)^2 \cdot F_V{}^2}$$

Figure 7.5: Inductive (left) and resistive (right) behavior in the hydraulic domain.

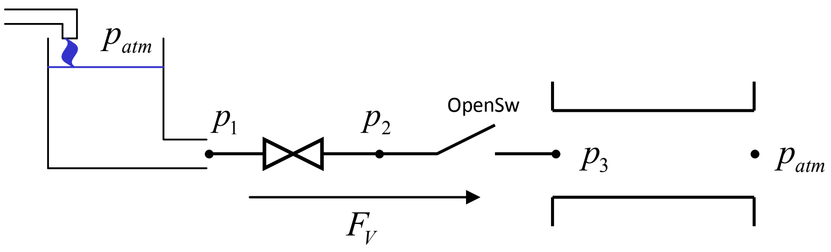


Figure 7.6: Diagram of the tank and drain pipe system.

$$p_1 = f(t)$$
$$p_1 - p_2 = \left(\frac{1}{S \cdot c_{desc}}\right)^2 F_V{}^2$$
$$p_2 - p_3 = 0$$
$$I \cdot \frac{dF_V}{dt} = p_3 - p_{atm}$$

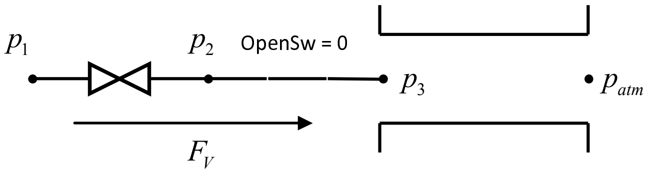


Figure 7.7: Drain pipe while the ideal switch is closed.

$$p_1 = f(t)$$
$$p_1 - p_2 = \left(\frac{1}{S \cdot c_{desc}}\right)^2 F_V{}^2$$
$$F_V = 0$$
$$I \cdot \frac{dF_V}{dt} = p_3 - p_{atm}$$

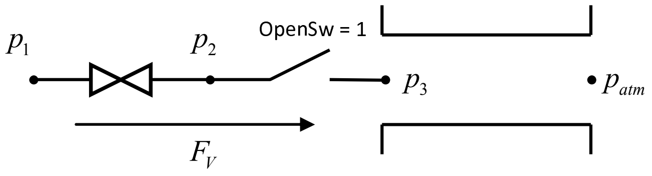


Figure 7.8: Drain pipe while the ideal switch is open.

Let's consider the hydraulic circuit shown in Figure 7.6. The liquid stored in the tank is drained through a valve-regulated pipe that is modeled as a resistive component, an ideal switch, and an inductive component. It is assumed that the pressure at the tank bottom is a known function of time, $p_1 = f(t)$, and the atmospheric pressure (p_{atm}) is a known constant.

The model equations depend on the switch mode, as described in Figures 7.7 and 7.8. Observe that the number of DoF depends on the switch mode.

– While the ideal switch is closed, the model has one DoF. The volumetric flow rate (F_V) can be selected as state variable.

– While the ideal switch is open, the model has zero DoF. The volumetric flow rate (F_V) is an algebraic variable that is calculated from the equation $F_V = 0$. The DAE index can be reduced by differentiating this equation (i.e., $F_V = 0$) and adding the obtained equation ($\frac{dF_V}{dt} = 0$) to the system.

At the time of writing this book, the Modelica modeling environments don't support the simulation of models with variable number of DoF. The model developer needs to modify the modeling hypotheses, in order to avoid runtime changes in the number of DoF.

An approach consists in describing the switch as a resistive component with low resistance while closed, and high resistance while open. In this way, the switch's constitutive relationship contains both the volumetric flow rate and the pressure drop in the two modes. If the ideal switch is replaced by the resistive switch, the drain pipe model depicted in Figure 7.6 has one DoF with independence of the switch mode.

7.5.4 Tanks connected in parallel

In the example discussed in Section 7.5.3, the high-index problem arises because the constitutive relationship of the open ideal switch ($F_V = 0$) only contains F_V (having, in consequence, F_V to be calculated from this equation), and F_V appears differentiated in the constitutive relationship of the hydraulic inductor.

The dual problem arises, for instance, when two storage tanks are connected in parallel to a liquid source, so that the flow to one of the tanks is regulated by an ideal switch. The tank is described as a hydraulic capacitor that stores potential energy. The number of DoF of this model depends on the switch mode, as shown in

Figures 7.9 and 7.10. While the ideal switch is closed, its constitutive relationship only contains the pressures at the switch connectors. As these variables appear differentiated in the tank's constitutive relationships, it is a high-index DAE system.

7.5.5 Resistive switch

As commented in Section 7.5.3, the difficulties associated to the use of ideal switches are avoided using resistive switches. This is an approach commonly adopted by model developers.

A model of a resistive switch is shown in Figure 7.11. It has two modes: open and closed. The two constitutive relationships (one per mode) are algebraic equations that contain the connector pressures and the volumetric flow rate. The resistive switch has low resistance while closed and high resistance while open.

The $\varepsilon\,()$ function that appears in the constitutive relationships of Figure 7.11 is any function that returns a value small enough to be considered negligible. A different function is employed in each mode. A possible selection is:

$$p_A - p_B = \varepsilon \cdot F_V \qquad \text{if} \quad OpenSw = 0 \tag{7.101}$$

$$p_A - p_B = \frac{1}{\varepsilon} \cdot F_V \qquad \text{if} \quad OpenSw = 1 \tag{7.102}$$

where in this case ε is a parameter small enough for the term in which intervenes to be negligible. As shown in Figures 7.12 – 7.15, this resistive switch produces models with the same number of DoF in the open and closed modes.

$$\frac{S_1}{\rho\cdot g}\cdot\frac{dp_1}{dt}=F_{V,1}$$
$$\frac{S_2}{\rho\cdot g}\cdot\frac{dp_2}{dt}=F_{V,2}$$
$$F_{V,2}=0$$
$$F_{V,S}=F_{V,1}+F_{V,2}$$
$$F_{V,S}=f(t)$$

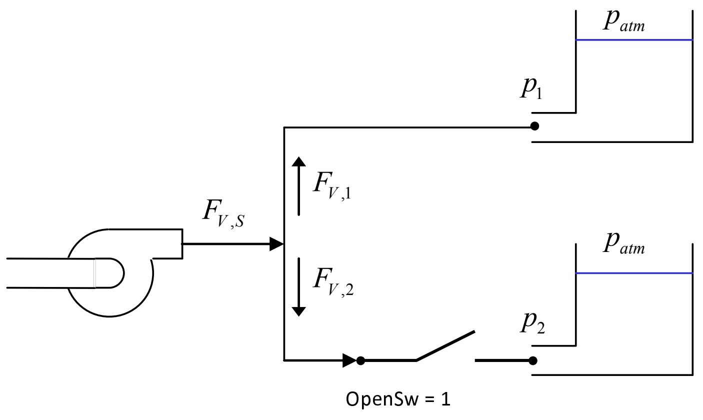


Figure 7.9: Ideal switch is open. The model has two DoF.

$$\frac{S_1}{\rho\cdot g}\cdot\frac{dp_1}{dt}=F_{V,1}$$
$$\frac{S_2}{\rho\cdot g}\cdot\frac{dp_2}{dt}=F_{V,2}$$
$$p_1=p_2$$
$$F_{V,S}=F_{V,1}+F_{V,2}$$
$$F_{V,S}=f(t)$$

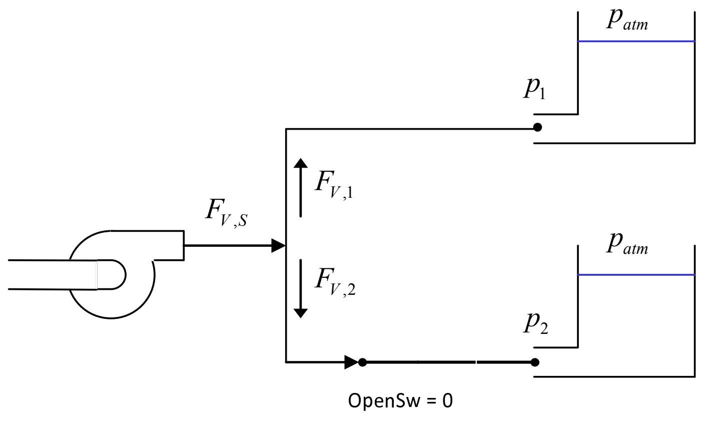
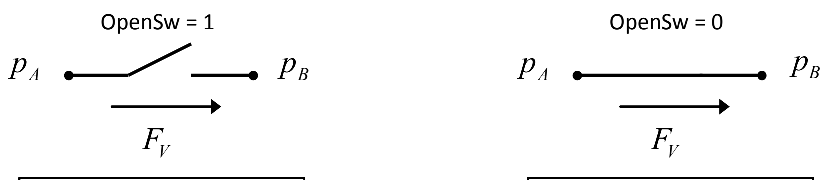


Figure 7.10: Ideal switch is closed. The model has one DoF.



$$\boxed{F_V+\varepsilon(p_A,p_B)=0}\qquad\boxed{p_A-p_B+\varepsilon(F_V)=0}$$

Figure 7.11: Resistive switch in the hydraulic domain.

$$\begin{aligned}
[p_1] &= f(t)\\
p_1 - [p_2] &= \left(\frac{1}{S\cdot c_{desc}}\right)^2 \cdot F_V^{\,2}\\
p_2 - [p_3] &= \frac{1}{\varepsilon}\cdot F_V\\
I\cdot\left[\frac{dF_V}{dt}\right] &= p_3 - p_{atm}
\end{aligned}$$

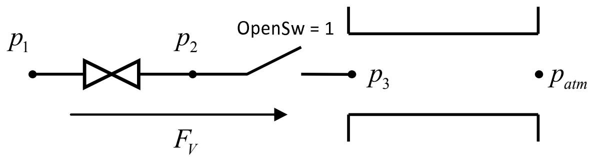


Figure 7.12: Drain pipe with resistive switch open. F_V is state variable.

$$\begin{aligned}
[p_1] &= f(t)\\
p_1 - [p_2] &= \left(\frac{1}{S\cdot c_{desc}}\right)^2 \cdot F_V^{\,2}\\
p_2 - [p_3] &= \varepsilon\cdot F_V\\
I\cdot\left[\frac{dF_V}{dt}\right] &= p_3 - p_{atm}
\end{aligned}$$

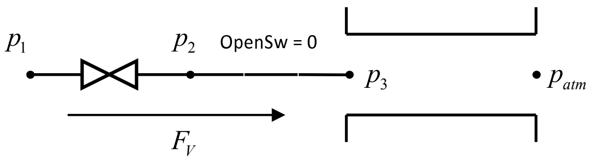


Figure 7.13: Drain pipe with resistive switch closed. F_V is state variable.

$$\begin{aligned}
[F_{V,S}] &= f(t)\\
p_1 - p_2 &= \frac{1}{\varepsilon}\cdot[F_{V,2}]\\
F_{V,S} &= [F_{V,1}] + F_{V,2}\\
\frac{S_1}{\rho\cdot g}\cdot\left[\frac{dp_1}{dt}\right] &= F_{V,1}\\
\frac{S_2}{\rho\cdot g}\cdot\left[\frac{dp_2}{dt}\right] &= F_{V,2}
\end{aligned}$$

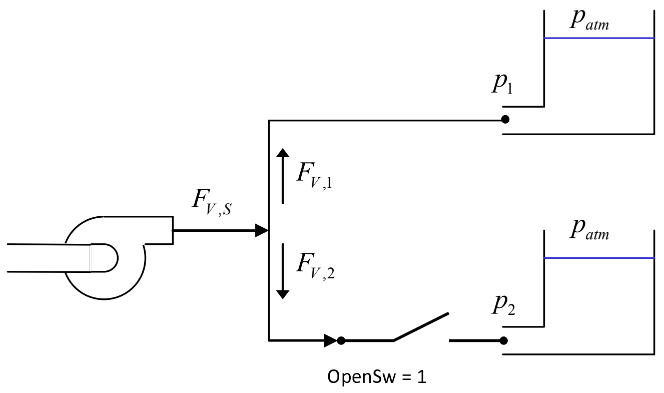


Figure 7.14: Tanks connected in parallel with resistive switch open. p_1 and p_2 are state variables.

$$\begin{aligned}
[F_{V,S}] &= f(t)\\
p_1 - p_2 &= \varepsilon\cdot[F_{V,2}]\\
F_{V,S} &= [F_{V,1}] + F_{V,2}\\
\frac{S_1}{\rho\cdot g}\cdot\left[\frac{dp_1}{dt}\right] &= F_{V,1}\\
\frac{S_2}{\rho\cdot g}\cdot\left[\frac{dp_2}{dt}\right] &= F_{V,2}
\end{aligned}$$

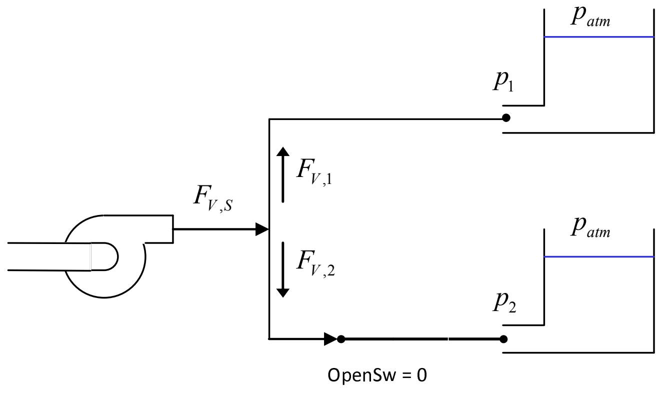


Figure 7.15: Tanks connected in parallel with resistive switch closed. p_1 and p_2 are state variables.

7.6 Model initialization

The model initialization, this is, the calculation of the model variables at the initial time, is problematic in some cases. If the model contains nonlinear algebraic loops, these are solved by applying iterative methods, using as initial guesses the corresponding values provided by the model developer. Depending on the problem in particular and on the proximity of the initial guess to the actual solution, the iterative method may not converge.

Once the model has been solved at the initial time, the solution of the algebraic loops is much less problematic during the integration of the continuous-time problem. The initial guess for iterating the algebraic loop is the value of the variable calculated at the previous time step. If the iterative method does not converge, the length of the integration time step can be reduced, so that the searched solution gets closer to the initial guess.

Observe that when an event produces instantaneous changes in the model state, the situation can be as problematic as solving the initialization problem. The values of the variables before the event are used as initial guess for Newton's iteration. The abrupt change in the model state can make the actual solution of the algebraic loop to be "too far" from the initial guess, so that the iteration of the restart problem does not converge.

The features provided by the Modelica language for defining the initialization problem are discussed in this section. These are based on supporting separated definition of the initialization problem and the dynamic problem, so that they can be defined using different sets of equations and, consequently, their computational causality is analyzed separately.

7.6.1 Posing the initialization problem

The initialization problem consists in calculating consistent values for all the model variables at the initial time. The vector of unknown variables to evaluate is:

$$\{\mathrm{der}\left(\mathbf{x}\right), \mathbf{x}, \mathbf{y}, \mathbf{q}, \mathrm{pre}\left(\mathbf{q}\right), \mathbf{p}, \mathbf{c}\} \tag{7.103}$$

where the vector components represent the derivatives ($\mathrm{der}(\mathbf{x})$), the variables that appear differentiated ($\mathbf{x}$), the algebraic continuous-time variables ($\mathbf{y}$), the discrete-

```
model Example1
  parameter Real x0=1;
  parameter Real a=2;
  parameter Real b=3;
  parameter Boolean steadyState=true;
  Real x(start=x0, fixed= not steadyState);
equation
  der(x) = a*x + b;
initial equation
  if steadyState then
    der(x) = 0;
  end if;
end Example1;
```

Modelica Code 7.1: Initialization of a continuous-time model.

time variables ($\mathbf{q}$), the "previous" value of the discrete time variables ($\text{pre}(\mathbf{q})$), the parameters ($\mathbf{p}$), and the Boolean conditions of the if and when clauses $\mathbf{c}$.

These unknown variables are calculated by solving the equations and algorithms that describe the continuous-time behavior of the model, and a set of additional constraints named **initial conditions**. The number of these initial conditions has to be equal to the number of continuous-time state variables ($\leq \dim(\mathbf{x})$), plus the number of parameters ($= \dim(\mathbf{p})$), plus the number of discrete-time variables ($= \dim(\mathbf{q})$).

7.6.2 Continuous-time variables

The initial value of the continuous-time variables can be specified using the **start** and **fixed** attributes. For instance, declaring the x variable as

```
Real x (start = x0, fixed = true);
```

these values of the *start* and *fixed* attributes are translated by the modeling environment into the following initial condition for the x variable:

```
x = x0
```

If the *fixed* attribute is not specified, its by-default value is:

`fixed = true`	for constants and parameters
`fixed = false`	for other variables

Another way of specifying initial conditions is by means of the **initial equation** and **initial algorithm** sections. For instance, consider the Modelica Code 7.1. In this model, the set of equations that defines the initialization problem depends on the value of the Boolean `steadyState` parameter. In consequence, the modeling environment needs to be able to calculate the value of this parameter before analyzing the computational causality of the model, and the model developer will not be allowed to modify the value of this parameter after the model translation.

If `steadyState` equals *true*, then the following equation is an initial condition:

```
der(x) = 0
```

If `steadyState` equals *false*, then the initial condition is:

```
x = x0
```

Additional initial conditions are the values assigned to the parameters `a` and `b`, this is, `a = 2`, `b = 3`.

The equations and algorithms that define the continuous-time behavior of the model have to be satisfied also at the model initialization. The continuous-time behavior of the model shown in Modelica Code 7.1 is described using only one equation, which has to be satisfied at the initial time:

```
der(x) = a*x + b
```

Summarizing the previous discussion, the initialization problem depends on the value assigned to the `steadyState` parameter. Depending on this value, the initialization problem is posed as described on the left or right column of the following table.

`steadyState = true`	`steadyState = false`
`a = 2, b = 3`	`a = 2, b = 3`
`der(x) = a*x + b`	`der(x) = a*x + b`
`der(x) = 0`	`x = x0`

In both cases, the problem is well posed. It consists of 5 equations to calculate 5 unknown variables: `steadyState`, `a`, `b`, `x` and `der(x)`.

7.6.3 Simple plane pendulum

The model of a pendulum is used in this section to show different ways of specifying the initialization problem. Let's consider the pendulum depicted in Figure 7.16. The pendulum length (L) and mass (m) are time-independent quantities.

The gravitational acceleration is $g = 9.81$ m·s^{-2}. The angle with respect to the vertical (φ) and the angular velocity (w) are described by the following equations:

$$\dot{\varphi} = w \tag{7.104}$$
$$J \cdot \dot{w} = -m \cdot g \cdot L \cdot \sin(\varphi) \tag{7.105}$$

The Cartesian coordinates (x, y) are related with the angle (φ) and length (L):

$$x = L \cdot \sin(\varphi) \tag{7.106}$$
$$y = L \cdot \cos(\varphi) \tag{7.107}$$

Three different ways of initializing the pendulum model are shown in Modelica Code 7.2.

1. Initial values are assigned to the parameters and the variables that appear differentiated ($\varphi(0) = 1$ rad, $w(0) = 0$ rad·s^{-1}).
2. Initial values are assigned to the vertical Cartesian coordinate ($y(0) = 0.9$ m) and the angular velocity ($w(0) = 0$ rad·s^{-1}). As two different values of $\varphi(0)$ may correspond to the same value of $y(0)$, an initial guess for the iterative calculation of $\varphi(0)$ is provided: $\varphi(0) \simeq 0.1$ rad.
3. Initial values are assigned to the Cartesian coordinates ($x(0) = 0.5$ m, $y(0) = 0.9$ m) and the angular velocity ($w(0) = 0$ rad·s^{-1}). The length of the pendulum (L) and the initial value of the angle ($\varphi(0)$) are not specified, but initial guesses are provided for their iterative calculation: $L \simeq 1$ m, $\varphi(0) \simeq 0.1$ rad.

7.6.4 Discrete-time variables

The **"previous" value of the discrete-time variables** can be initialized in the following two ways:

1. Setting the value of the **start** and **fixed** attributes. For instance, the following declarations

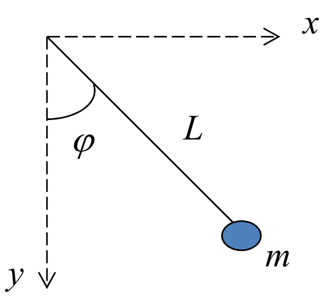


Figure 7.16: Simple plane pendulum.

```
model pendulum
   parameter Real m=1, g=9.81;
   parameter Real L=1;
   parameter Real J=m*L^2;
   Real  phi (start=1, fixed=true);
   Real  w   (start=0, fixed=true);
   Real  x;
   Real  y;
equation
   der(phi) = w;
   J*der(w) = -m*g*L*sin(phi);
   x = sin(phi)*L;
   y = cos(phi)*L;
end pendulum;
```

```
model pendulum
   parameter Real m=1, g=9.81;
   parameter Real L=1;
   parameter Real J=m*L^2;
   Real  phi (start=0.1, fixed=false);
   Real  w   (start=0,   fixed=true);
   Real  x;
   Real  y   (start=0.9, fixed=true);
equation
   der(phi) = w;
   J*der(w) = -m*g*L*sin(phi);
   x = sin(phi)*L;
   y = cos(phi)*L;
end pendulum;
```

```
model pendulum
   parameter Real m=1, g=9.81;
   parameter Real L (fixed=false) = 1;
   parameter Real J=m*L^2;
   Real  phi (start=0.1, fixed=false);
   Real  w   (start=0,   fixed=true);
   Real  x   (start=0.5, fixed=true);
   Real  y   (start=0.9, fixed=true);
equation
   der(phi) = w;
   J*der(w) = -m*g*L*sin(phi);
   x = sin(phi)*L;
   y = cos(phi)*L;
end pendulum;
```

Modelica Code 7.2: Three different ways of initializing the pendulum model.

```
Boolean b (start = false, fixed = true);
Integer i (start = 1    , fixed = true);
```

produce the two following initial conditions:

```
pre(b) = false;
pre(i) = 1;
```

2. Writing the initial conditions within **initial equation** and **initial algorithm** sections. For instance,

```
initial equation
   pre(xd) = 0;
   pre(u)  = 0;
```

The initial conditions on the **value of the discrete-time variables** have to be written within when clauses whose activation condition is the **initial()** function. For instance:

```
equation
   when { initial(), condition1, ...} then
      v = ...
   end when;
```

Observe that a when clause is active during the initialization if and only if the clause condition is the initial() function. It is not possible to specify initial conditions using other conditions, such as `not time < 0`, or `time >= 0`.

If an initial condition is not specified for a discrete-time variable, it is then by-default assumed that the value of the variable is equal to its previous value. For instance, if the when clause where the `v` variable is evaluated does not contain the initial() function as a trigger condition, then the following initial condition is assumed: `v = pre(v)`.

7.6.5 Control loop

The model of the control loop shown in Figure 7.17 allows to illustrate the initialization of the discrete-time variables. The control loop is represented as a block diagram. The circle indicates the subtraction operation performed to calculate the

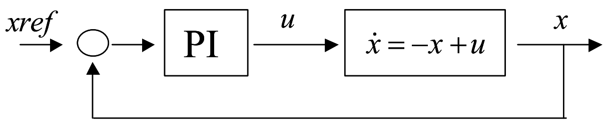


Figure 7.17: Control loop.

difference between the setpoint value (x_{ref}), and the actual output of the plant (x). The result of this operation, which is named *error signal* ($= x_{ref} - x$), is the input to the PI controller. The output of the PI controller (u) is the input to the plant.

The controller is a discrete-time PI controller, with sampling period T_S, that is described by the following difference equations:

$$\begin{aligned} x_d &= \mathbf{pre}\,(x_d) + \frac{T_S}{T} \cdot (x_{ref} - x) & (7.108) \\ u &= k \cdot (x_{ref} - x) + x_d & (7.109) \end{aligned}$$

where $x_{ref} - x$ is the error signal, k and T are the proportional and integral parameters respectively, x_d is the integral term, which is calculated from Eq. (7.108), and $k \cdot (x_{ref} - x)$ is the proportional term. As described by Eq. (7.109), the controller output (u) is calculated by adding the proportional and integral terms.

The plant is described by the following equation:

$$\dot{x} = -x + u \qquad (7.110)$$

Four different ways of initializing this model are described next. In each case, the fragment of Modelica code and the associated initialization problem are shown.

Case 1

Let's suppose that the model of the control loop shown in Figure 7.17 is described as indicated in the following fragment of code:

```
model ControlLoop
   parameter Real k=10, T=1;  // PI controller parameters
   parameter Real Ts=0.01   "Sampling period";
             Real xref      "Reference";
             Real x  (fixed=true, start=2);
   discrete  Real xd (fixed=true, start=0);
   discrete  Real u  (fixed=true, start=0);
equation
   // Reference
   xref = sin(time);
   // Plant
   der(x) = -x + u;
   // Discrete PI controller
   when sample(Ts,Ts) then
      xd = pre(xd) + Ts/T*(xref-x);
      u  = k*(xref-x)+xd;
   end when;
end ControlLoop;
```

The unknown variables of the initialization problem are:

```
   k, T, Ts
   xref
   x, der(x)
   xd, u, pre(xd), pre(u)
```

The when clause is not active during the initialization. Therefore, the following equations are added to the initialization problem:

```
   xd := pre(xd)
   u  := pre(u)
```

The sorted and solved equations of the initialization problem are:

```
   k := 10, T := 1, Ts := 0.01
   xref    := sin(0)    // = 0
   x       := x.start   // = 2
   pre(xd) := xd.start  // = 0
   pre(u)  := u.start   // = 0
   xd      := pre(xd)   // = 0
   u       := pre(u)    // = 0
   der(x)  := -x+u      // = -2
```

Case 2

Let's consider another initial conditions for the control loop model:

```
model ControlLoop
   parameter Real k=10, T=1;  // PI controller parameters
   parameter Real Ts=0.01   "Sampling period";
             Real xref      "Reference";
             Real x  (fixed=true, start=2);
   discrete  Real xd (fixed=true, start=0);
   discrete  Real u  (fixed=true, start=0);
equation
   // Reference
   xref = sin(time);
   // Plant
   der(x) = -x + u;
   // Discrete PI controller
   when {initial(),sample(Ts,Ts)} then
      xd = pre(xd) + Ts/T*(xref-x);
      u  = k*(xref-x)+xd;
   end when;
end ControlLoop;
```

As the initial() function is a condition of the when clause, the instantaneous equations of the when clause are added to the initialization problem.

```
      xd := pre(xd) + Ts/T*(xref-x)
      u  := k*(xref-x) + xd
```

The sorted and solved equations of the initialization problem are:

```
      k := 10, T := 1, Ts := 0.01
      xref    := sin(0)    // = 0
      x       := x.start   // = 2
      pre(xd) := xd.start  // = 0
      pre(u)  := u.start   // = 0
      xd      := pre(xd)+Ts/T*(xref-x) // = -0.02
      u       := k*(xref-x)+xd         // = -20.02
      der(x)  := -x+u                  // = -22.02
```

Case 3

Let's suppose that the control loop model is initialized as shown below. In this case, the when clause is active during the initialization, and the previous values of the discrete-time variables are assigned in the initial equation section.

```
model ControlLoop
   parameter Real k=10, T=1;  // PI controller parameters
   parameter Real Ts=0.01   "Sampling period";
             Real xref      "Reference";
             Real x  (fixed=true, start=2);
   discrete  Real xd;
   discrete  Real u;
equation
   // Reference
   xref = sin(time);
   // Plant
   der(x) = -x + u;
   // Discrete PI controller
   when {initial(),sample(Ts,Ts)} then
      xd = pre(xd) + Ts/T*(xref-x);
      u  = k*(xref-x)+xd;
   end when;
initial equation
   pre(xd) = 0;
   pre(u)  = 0;
end ControlLoop;
```

The sorted and solved equations of the initialization problem are:

```
      k := 10, T := 1, Ts := 0.01
      xref    := sin(0)    // = 0
      x       := x.start   // = 2
      pre(xd) := 0         // = 0
      pre(u)  := 0         // = 0
      xd      := pre(xd)+Ts/T*(xref-x)  // = -0.02
      u       := k*(xref-x)+xd          // = -20.02
      der(x)  := -x+u                   // = -22.02
```

Case 4

In this case, the control loop model is initialized imposing that it is at steady state:

```
model ControlLoop
   parameter Real k=10, T=1;  // PI controller parameters
   parameter Real Ts=0.01   "Sampling period";
             Real xref      "Reference";
             Real x  (fixed=false, start=2);
   discrete  Real xd;
   discrete  Real u;
equation
   // Reference
   xref = sin(time);
   // Plant
   der(x) = -x + u;
   // Discrete PI controller
   when {initial(),sample(Ts,Ts)} then
      xd = pre(xd) + Ts/T*(xref-x);
      u  = k*(xref-x)+xd;
   end when;
initial equation
   pre(xd) = xd;
   pre(u)  = u;
   der(x)  = 0;
end ControlLoop;
```

The sorted equations of the initialization problem are:

```
k := 10, T := 1, Ts := 0.01
xref    := sin(0)    // =0
der(x)  = 0
// Linear system of 4 simultaneous equations
// Unknown variables: xd, pre(xd), u, x
| pre(xd) = xd
| xd      = pre(xd)+Ts/T*(xref-x)
| u       = k*(xref-x)+xd
| der(x)  = -x+u
pre(u)  = u
```

The sorted and solved equations are:

```
k := 10, T := 1, Ts := 0.01
xref    := sin(0)    // = 0
der(x)  := 0         // = 0
xd      := xref      // = 0
u       := xref      // = 0
x       := xref      // = 0
pre(xd) := xd        // = 0
pre(u)  := u         // = 0
```

7.6.6 Initial model

The model developer can specify initial conditions using the *start* and *fixed* attributes, and the initial equation and initial algorithm sections. These initial conditions, the by-default initial conditions, and the equations and algorithms that describe the continuous-time behavior, constitute the initialization problem.

In posing the initialization problem, Modelica allows to replace some of these equations that describe the model continuous-time behavior, by other equations that are only used at the model initialization. To this end, if sentences with the **initial()** function as condition can be employed. For instance,

```
equation
   y = if initial() then
          // expression used only at the model initialization
       else
          // expression used during the simulation
```

This feature allows to employ a simplified model to describe the initialization problem (e.g., a model linearized at the initial operating point), with the aim of facilitating its numerical solution.

7.7 Further reading

The OHM formalism and the Omola algorithm for hybrid model simulation are described in (Andersson 1990) and (Andersson 1994). We have used these two theses as references in preparing this lesson.

The event detection and iteration are discussed in (Cellier 1979) and (Cellier et al. 1993).

The initialization procedure for Modelica models is described in (Mattsson et al. 2002). The explanations given in Section 7.6 are extracted from this article.

Simulation of variable structure models is discussed in (Elmqvist 1993), (Elmqvist et al. 1993) and (Cellier et al. 1995). We have employed these three articles as references in preparing this lesson.

The Modelica features for hybrid modeling are described in (ModelicaTM 2000), (Otter 2009), (Fritzson 2011) and (Tiller 2001). As their reading was recommended in previous lessons, probably the reader is now familiar with these books.

Lesson 8

Event detection and handling

8.1 Introduction
8.2 Simultaneous events
8.3 Crossing function
8.4 Determination of the event instant
8.5 Chattering
8.6 Further reading

Learning objectives

After studying the lesson, students should be able to:

- Discuss how simultaneous events are handled in Modelica.
- Discuss how event chains are executed by Modelica modeling environments such as Dymola and OpenModelica.
- Discuss the event detection mechanism based on crossing functions that is implemented in Modelica modeling environments such as Dymola and OpenModelica.
- Discuss how the trigger time of events is calculated by Modelica modeling environments such as Dymola and OpenModelica.
- Use the noEvent operator of Modelica.
- Detect and avoid chattering.

8.1 Introduction

In this section, the detection and handling of events are analyzed more in-depth. The concept of the crossing function is also explained, showing how the Modelica modeling environments use crossing functions to detect the events. Finally, the chattering is explained.

8.2 Simultaneous events

The execution of an event can generate the triggering, in that same instant, of another event. This happens when the solution of the restart problem does not satisfy one of the invariants, and, consequently, the event corresponding to this invariant is immediately executed. In this way, several events can be sequentially executed until all invariants are satisfied, and then, the solution of the continuous-time problem is resumed. The execution of a sequence of events is called an **event chain**.

When only one invariant is not satisfied, there is no doubt about how to proceed. However, several events can be detected simultaneously during the solution of the continuous-time problem. Also, several invariants can be not satisfied at certain step in the execution of an event chain. In both cases, it is necessary to establish a criterion to decide how to execute these simultaneously triggered events.

The order in which simultaneously triggered events are executed may be irrelevant. This is typically the case when these events affect to different parts of the model that don't interact among them. However, the execution order may be relevant, affecting to the solution of the restart problem. Let's consider, for instance, the model described by Eqs. (8.1) – (8.5).

$$\frac{dx_1}{dt} = 1 \tag{8.1}$$

$$\frac{dx_2}{dt} = -1 \tag{8.2}$$

$$x_1 + x_2 = y \tag{8.3}$$

$$\begin{array}{l}\textbf{when } x_1 \geq 0.5 \cdot y \textbf{ then}\\ \quad \textbf{reinit}\,(x_1, 0)\\ \textbf{end when};\end{array} \tag{8.4}$$

$$\begin{array}{l}\textbf{when } x_2 \leq 0.5 \cdot y \textbf{ then}\\ \quad \textbf{reinit}\,(x_2, 10)\\ \textbf{end when};\end{array} \tag{8.5}$$

The initial values assigned to the state variables are:

$$x_1(0) = 0 \qquad\qquad x_2(0) = 10 \tag{8.6}$$

Observe that, at time $t = 5$, it is satisfied

$$x_1(5) = 5 \qquad\qquad x_2(5) = 5 \qquad\qquad y(5) = 10 \tag{8.7}$$

and, consequently, the two events are triggered simultaneously.

If the event whose condition is $(x_1 \geq 0.5 \cdot y)$ is executed in first place, the results shown in Table 8.1 are obtained. As the event condition $(x_2 \leq 0.5 \cdot y)$ is *false*, the associated event is no longer triggered and, therefore, is not executed.

On the contrary, if the event with the condition $(x_2 \leq 0.5 \cdot y)$ is executed in the first place, the results shown in the Table 8.2 are obtained. In this case, the condition $(x_1 \geq 0.5 \cdot y)$ is *false*, so the associated event is no longer triggered.

Comparing the results shown in Tables 8.1 and 8.2, it can be seen that the model state after the event depends on the selection of which of the simultaneously triggered events is executed in the first place.

There are several methods, deterministic and stochastic, to decide the **execution order** of the events triggered simultaneously. One method consists in triggering the events according to the order in which they have been defined in the model code. The events scheduled in time can be sorted from the beginning, establishing a queue.

Let's suppose, for instance, that the events e_1 and e_2 have been triggered simultaneously. The event e_1 is executed first, because it has been defined before the event e_2. The solution of the restart problem satisfies one of the three following conditions:

1. If all the invariant expressions are satisfied, then the solution of the continuous-time problem is resumed.

2. If only one invariant expression is not satisfied, the event associated to this invariant is executed.

3. If several invariant expressions are not satisfied, the event with less order of definition, among the events associated to these invariant, is executed.

Summarizing the previous discussion, the algorithm employed to decide which event to execute would be:

Table 8.1: Result of executing the event whose condition is $(x_1 \geq 0.5 \cdot y)$.

Before the event execution	**After the event execution**
$x_1 = 5$	$x_1 = 0$
$x_2 = 5$	$x_2 = 5$
$y = 10$	$y = 5$
$(x_1 \geq 0.5 \cdot y) = true$	$(x_1 \geq 0.5 \cdot y) = false$
$(x_2 \leq 0.5 \cdot y) = true$	$(x_2 \leq 0.5 \cdot y) = false$

Table 8.2: Result of executing the event whose condition is $(x_2 \leq 0.5 \cdot y)$.

Before the event execution	**After the event execution**
$x_1 = 5$	$x_1 = 5$
$x_2 = 5$	$x_2 = 10$
$y = 10$	$y = 15$
$(x_1 \geq 0.5 \cdot y) = true$	$(x_1 \geq 0.5 \cdot y) = false$
$(x_2 \leq 0.5 \cdot y) = true$	$(x_2 \leq 0.5 \cdot y) = false$

Table 8.3: Result of executing the two events.

Before the event execution	**After the event execution**
$x_1 = 5$	$x_1 = 0$
$x_2 = 5$	$x_2 = 10$
$y = 10$	$y = 10$
$(x_1 \geq 0.5 \cdot y) = true$	$(x_1 \geq 0.5 \cdot y) = false$
$(x_2 \leq 0.5 \cdot y) = true$	$(x_2 \leq 0.5 \cdot y) = false$

```
model dispEvent
   Real x1(start=0, fixed=true);
   Real x2(start=10, fixed=true);
   Real y;
equation
   der(x1) = 1;
   der(x2) = -1;
   x1 + x2 = y;
   when x1 >= 0.5*y then
      reinit(x1,0);
   end when;
   when x2 <= 0.5*y then
      reinit(x2,10);
   end when;
end dispEvent;
```

Modelica Code 8.1: Model used to illustrate the simultaneous triggering of two events.

Algorithm:

Step 1. Execute the event with less order of definition among the triggered events. An event is triggered when the value of its invariant expression is *false*.

Step 2. Check whether there are triggered events. If this is the case, go to Step 1. Otherwise, resume the solution of the continuous-time problem.

Other different algorithm would be:

Algorithm:

Step 1. Determine and sort out, according to the definition order, the set of triggered events. This set is named E'.

Step 2. If the set E' is empty, resume the solution of the continuous-time problem, finishing this algorithm.

Step 3. Execute the first event of the E' set.

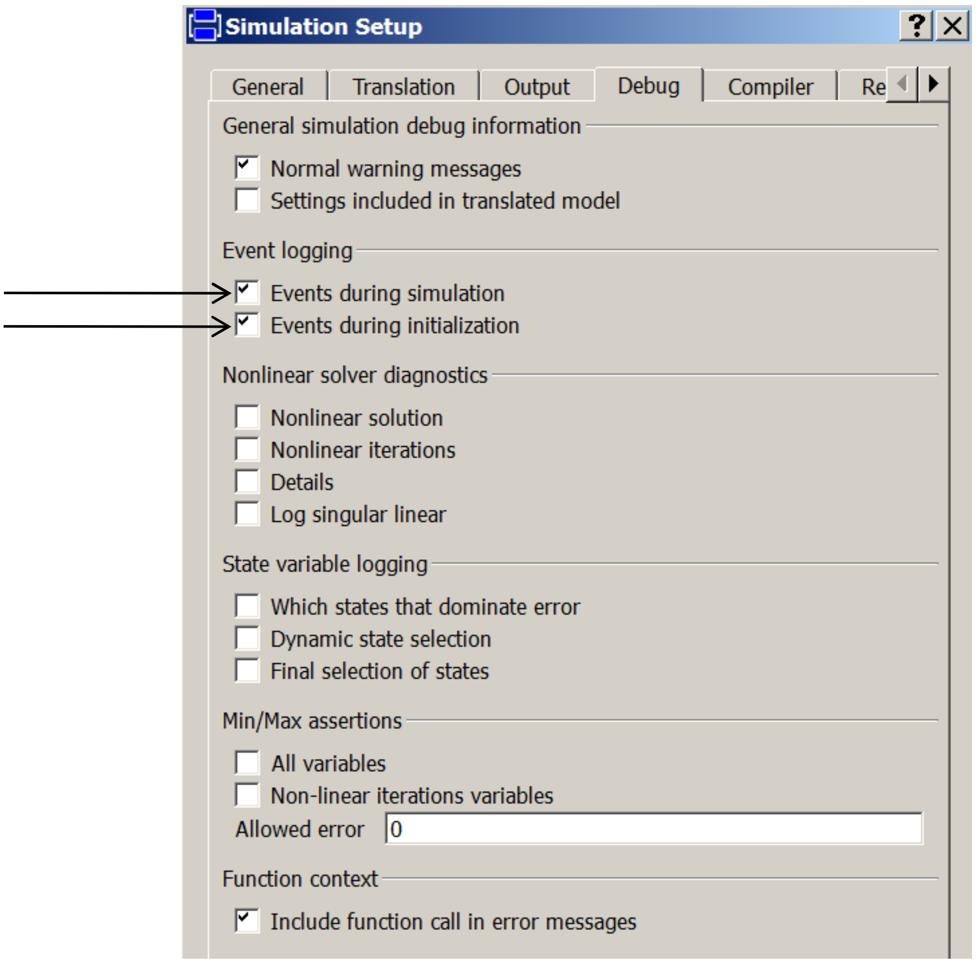
Step 4. Consider the next event of the sorted set E'. If this event has not been executed yet and it is still triggered, this event is executed.

Step 5. If every event of E' has been examined, go to Step 1. Otherwise, go to Step 4.

Another approach consists in executing simultaneously all the triggered events. This is the approach adopted in Modelica. To this end, the **single-assignment rule** is imposed. This rule states that all the instantaneous changes in a continuous-time or discrete-time state variable must be described in a single instantaneous equation. This guarantees that the same state variable is not changed by two instantaneous equations simultaneously active. The potential risk of executing simultaneously several events that assign different values to a same state variable is eliminated.

Applying this criterion (i.e., executing simultaneously all the triggered events) to the previous example, the result shown in the Table 8.3 is obtained. To check it, let's describe the model in Modelica as shown in Modelica Code 8.1, and simulate it using Dymola.

Dymola can be configured for writing information on the executed events in the message window during the simulation. To this end, the option *Event logging* has to be selected in the *Debug* tab of the *Simulation Setup* window, before translating the model (see Figure 8.1). The results obtained by executing the simulation during 16 s with Dymola are shown in Figure 8.2. Dymola writes in the message window the following report, describing the events detected and iterated (determination of the event trigger time) during the simulation.



General simulation debug information

- [✓] Normal warning messages
- [ ] Settings included in translated model

Event logging

- [✓] Events during simulation
- [✓] Events during initialization

Nonlinear solver diagnostics

- [ ] Nonlinear solution
- [ ] Nonlinear iterations
- [ ] Details
- [ ] Log singular linear

State variable logging

- [ ] Which states that dominate error
- [ ] Dynamic state selection
- [ ] Final selection of states

Min/Max assertions

- [ ] All variables
- [ ] Non-linear iterations variables

Allowed error 0

Function context

- [✓] Include function call in error messages

Figure 8.1: Configuring Dymola so that the information on the event detection and iteration during the model initialization and simulation is written in the message window.

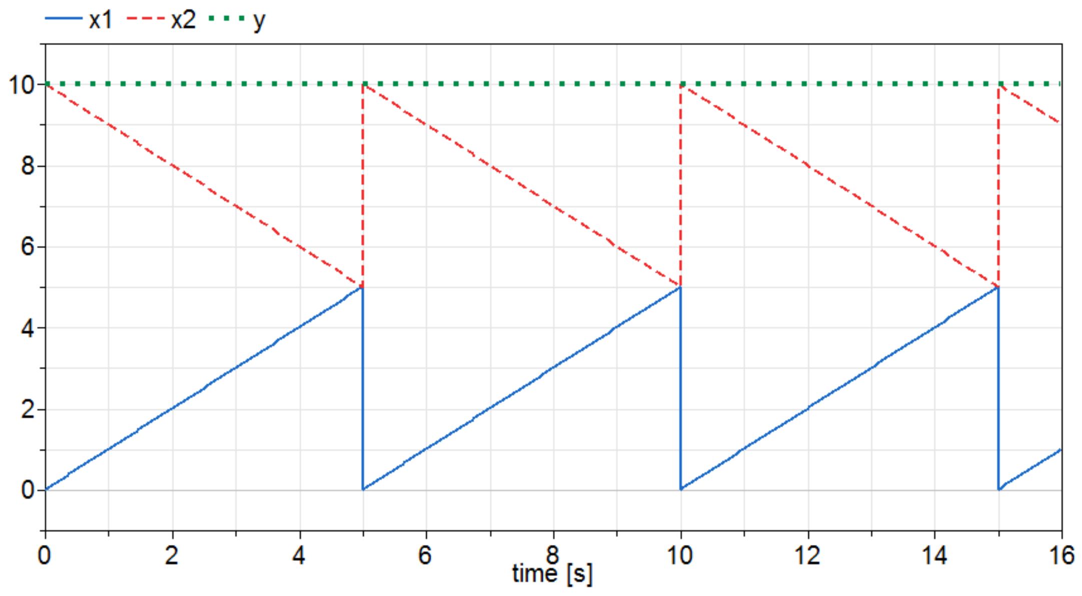


Figure 8.2: Simulation of Modelica Code 8.1.

```
Expression x2 <= 0.5*y became false ( (x2)-(0.5*y) = 5 )
Expression x1 >= 0.5*y became false ( (x1)-(0.5*y) = -5 )
Iterating to find consistent restart conditions.
      during event at Time :  0

Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dynasim))

Expression x2 <= 0.5*y became true ( (x2)-(0.5*y) = -9.99991e-011 )
Expression x1 >= 0.5*y became true ( (x1)-(0.5*y) = 1e-010 )
Iterating to find consistent restart conditions.
Expression x2 <= 0.5*y became false ( (x2)-(0.5*y) = 5 )
Expression x1 >= 0.5*y became false ( (x1)-(0.5*y) = -5 )
Iterating to find consistent restart conditions.
      during event at Time :  5.0000000001

Expression x2 <= 0.5*y became true ( (x2)-(0.5*y) = -5e-010 )
Expression x1 >= 0.5*y became true ( (x1)-(0.5*y) = 5e-010 )
Iterating to find consistent restart conditions.
Expression x2 <= 0.5*y became false ( (x2)-(0.5*y) = 5 )
Expression x1 >= 0.5*y became false ( (x1)-(0.5*y) = -5 )
Iterating to find consistent restart conditions.
      during event at Time :  10.0000000006

Expression x2 <= 0.5*y became true ( (x2)-(0.5*y) = -5.00001e-010 )
Expression x1 >= 0.5*y became true ( (x1)-(0.5*y) = 5e-010 )
Iterating to find consistent restart conditions.
Expression x2 <= 0.5*y became false ( (x2)-(0.5*y) = 5 )
Expression x1 >= 0.5*y became false ( (x1)-(0.5*y) = -5 )
Iterating to find consistent restart conditions.
      during event at Time :  15.0000000011

Integration terminated successfully at T = 16
```

The simulation of this hybrid model has been performed as follows.

- The initialization problem is solved.
- The numerical integration of the continuous-time problem starts, using as initial value the solution of the initialization problem.
- The numerical integration algorithm advances, using its own method for adjusting the time step size. The algorithm proceeds until it is detected that one or several event conditions have become *true*. The two event conditions of this model become true when time becomes greater that $t = 5$ s. As events have been detected, the numerical integration of the continuous-time problem is halted.

– Event iteration starts. The objective of this iterative algorithm is to calculate the precise time at which the events are triggered. The obtained result is a time interval that satisfies the following two conditions: (1) its length is smaller than a certain tolerance; (2) all the event conditions are *false* at the interval left-hand limit, and at least one is *true* at the right-hand limit. It is assumed that the event trigger time is the right-hand limit of this interval. In this case, it has the value 5.0000000001 s.

– The restart problem is solved at the calculated event trigger time. If the solution of the restart problem triggers events, these are executed. In this way, the event chain is executed until all the event conditions are *false*. There is no event chain in this model: the event conditions calculated solving the restart problem are *false*.

– The numerical integration of the continuous-time problem is resumed, employing as initial value the solution of the restart problem. If a variable step-size method is used (e.g., DASSL), the step size is reset.

– The numerical integration of the continuous-time problem proceeds. The event conditions are watched. When one or several event conditions become *true*, the integration is halted. This happens when time becomes greater that $t = 10$ s. Again, the event trigger time is found (event iteration), the restart problem is solved, and the numerical integration of the continuous-time problem is resumed.

– Events are detected when time becomes greater than $t = 15$ s. The same procedure is applied.

– The ending condition is satisfied when time reaches the value $t = 16$ s, therefore, the simulation finishes.

8.3 Crossing function

Modeling environments of hybrid systems typically use crossing functions for detecting events. To this end, the event conditions are automatically translated into crossing functions, which are watched during the continuous-time problem solution.

A **crossing function** is an expression whose result is positive while the event condition is *true*, and negative while it is *false*. Therefore, the crossing function crosses the zero value at the time instant in which the event condition changes its value (from *true* to *false*, or vice-versa).

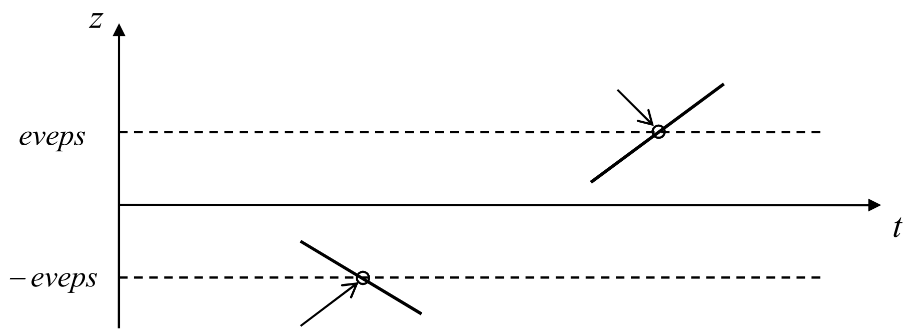


Figure 8.3: When z crosses $-eveps$ with negative slope, the event condition is considered to change its value from *true* to *false*. When z crosses $+eveps$ with positive slope, the event condition is considered to change its value from *false* to *true*.

Let's see an example. Suppose that the y variable is described by means of the following if sentence,

$$y = \textbf{if } x > xLimit \textbf{ then } y1 \textbf{ else } y2; \tag{8.8}$$

This sentence states that while $x > xLimit$, the y variable is equal to the $y1$ variable. Otherwise, the y variable is equal to the $y2$ variable. The event detection can be made by associating the event condition $x > xLimit$ to the following crossing function (named z):

$$z = x - xLimit \tag{8.9}$$

The Modelica modeling environments typically perform the event detection as follows (see Figure 8.3). A small interval around zero, $(-eveps, eveps)$, is defined.

– The change of the event condition from *false* to *true* is detected when z crosses *eveps* with positive slope.

– The change of the event condition from the *true* to *false* is detected when z crosses $-eveps$ with negative slope.

This is equivalent to associate to the event condition the two crossing functions shown below, and to watch their cross through the zero value.

$$zp = z + eveps \tag{8.10}$$
$$zn = z - eveps \tag{8.11}$$

If the z crossing function initially remains inside the interval $(-eveps, eveps)$ due to the initial conditions, the value of the crossing function is assumed to be zero during this time and it is used the corresponding branch of the `if` expression.

The value of *eveps* is very small. The by-default value used by Dymola and OpenModelica is 1E-10. This value can be changed, although this is not recommended: it is preferable to re-scale the model than modify *eveps*. In any case, the procedure to change *eveps* in Dymola is explained below.

To change the *eveps* value, the experiment file generated by Dymola (*dsin.txt*) has to be manually edited, after the model has been translated and before running the simulation. For avoiding Dymola to re-write the *dsin.txt* file when the simulation is launched, the simulation must not be executed from the user-interface of Dymola. Instead, it has to be launched by executing the following command from the operating system shell, in the working directory (where Dymola has generated the *dymosim.exe* and *dsin.txt* files):

```
dymosim dsin.txt
```

When the simulation finishes, Dymola stores the results in a file named *dsres.mat*. This file can be opened from the graphical-user-interface of Dymola (*Plot* > *Open Result*) to represent graphically the model variables.

To illustrate how Dymola performs the event detection, we are going to simulate the model shown in Modelica Code 8.2 using two different values of *eveps*. The first simulation of the model has been performed using the by-default value of *eveps*, i.e., 1E-10. In the second simulation, the value 0.6 has been manually assigned to *eveps*. To this end, the following line of the *dsin.txt* file has been replaced

```
1.0000000000000000E-010 # eveps  Hysteresis epsilon at event points
```

by:

```
0.6000000000000000      # eveps  Hysteresis epsilon at event points
```

and then *dymosim.exe* has been executed from a MS-DOS shell. Afterwards, the results have been loaded from Dymola to obtain a graphical representation.

The results of the two simulations are shown in Figure 8.4. The event condition $(x > 0)$ is translated into two crossing functions: $zp = x + eveps$, $zn = x - eveps$. The event condition is detected to change from *false* to *true* when x crosses $+eveps$ with positive slope, and from *true* to *false* when x crosses $-eveps$ with negative slope. This is clearly visible in the plot placed in the lower part of Figure 8.4.

```
model ejemEveps
   Real x;
   Real y;
equation
   y = if x > 0 then 1 else -1;
   x = sin(time);
end ejemEveps;
```

Modelica Code 8.2: Model to illustrate the event detection in Dymola.

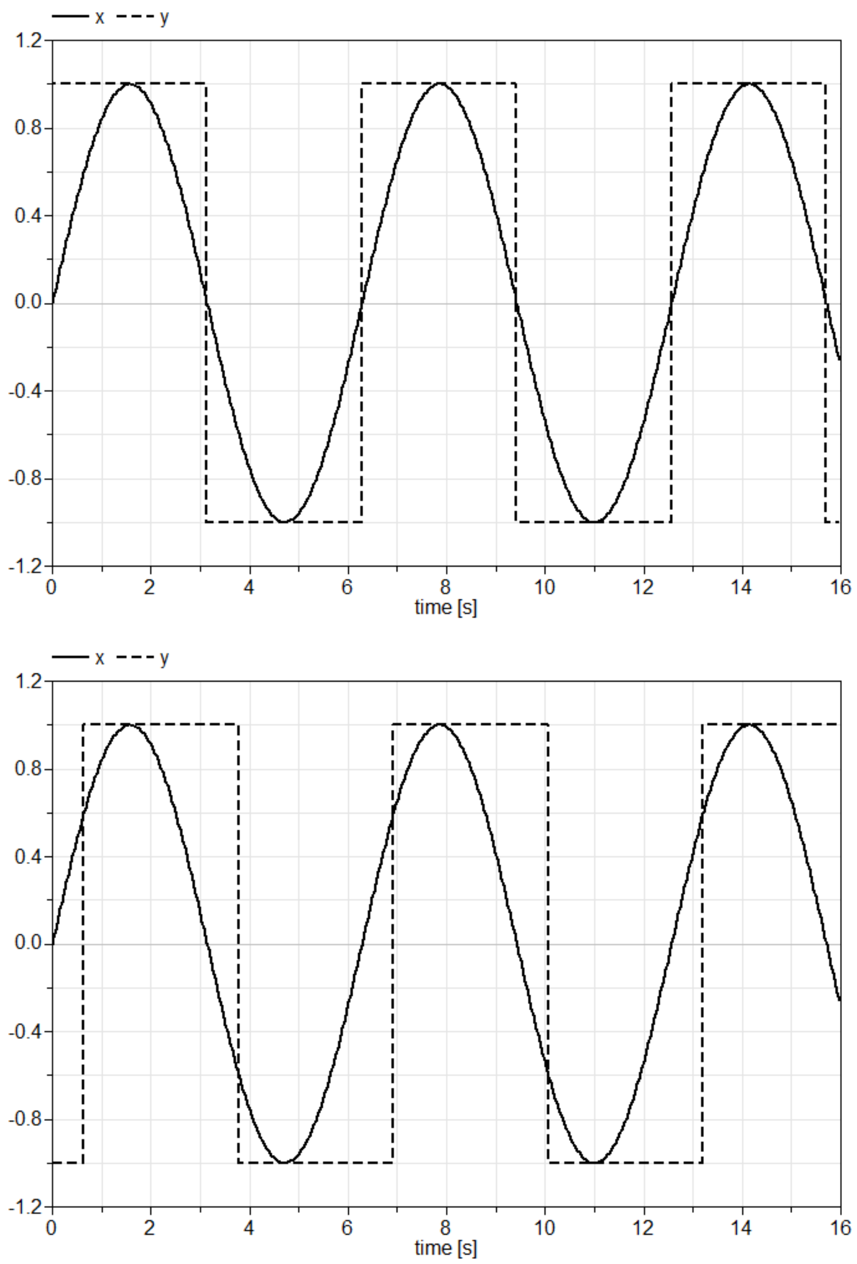


Figure 8.4: Simulation results for two values of *eveps*: 1E-10 (upper plot) and 0.6 (lower plot).

The mechanism used by the modeling environment to detect events is relevant in a practical sense, and must be known by the model developer, because this mechanism may result in numerical artifacts that condition the simulation result. An example is provided below.

Let's consider again the model of the ball that falls downwards due to gravity and bounces when hitting the floor. The ball is initially at rest, and it is dropped from a height of 10 m. The model is shown in Modelica Code 8.3.

Let's execute the simulation during 14 s. The result is shown in Figure 8.5. At the beginning of the simulation, the ball falls downwards accelerated by the gravity. When the ball hits the floor, the speed direction changes, going up and then falling down again. As the ball losses speed (an consequently, energy) with each bounce, the maximum height of the ball decreases between consecutive bounces.

Observe in Figure 8.5 that the ball falls below the floor level when the simulation time is around 13 s. This behavior is not described in the model: is a numerical artifact due to the mechanism used to detect the events.

To look into the reasons behind the falling of the ball below the floor level, let's repeat the simulation of the model with an initial height of the ball of $5E-9$ m. As in the previous simulation, the ball is initially at rest. We establish also 0.0003 s as the final time of the simulation.

Executing the simulation, the result shown in the upper plot of Figure 8.6 is obtained. If we zoom the interval $2.16E-4 < t < 2.52E-4$, it is noted (see the lower plot of the figure) that there comes a time when the ball bounces at the height $-eveps$ with so low energy that is not able to go beyond $eveps$. Consequently, Dymola does not detect that the condition $x \leq 0$ becomes *false*. For this reason, the event condition remains *true* from the bouncing performed at the time $2.36E-4$ s approximately. As the instantaneous equations of the when clause are activated when the condition changes from *false* to *true*, the when condition is not activated again. And, therefore, the speed is not reinitialized and the ball falls below the floor level.

```
model BouncingBall1
   Modelica.SIunits.Distance x(start=10, fixed=true);
   Modelica.SIunits.Velocity v(start=0, fixed=true);
   parameter Real c = 0.8 "Coeff. elastic bouncing";
   parameter Modelica.SIunits.Acceleration g = 9.8;
equation
   der(v) = -g;
   der(x) = v;
   when x <= 0 then
      reinit(v, -c*v);
   end when;
end BouncingBall1;
```

Modelica Code 8.3: Vertical fall and bouncing of a ball.

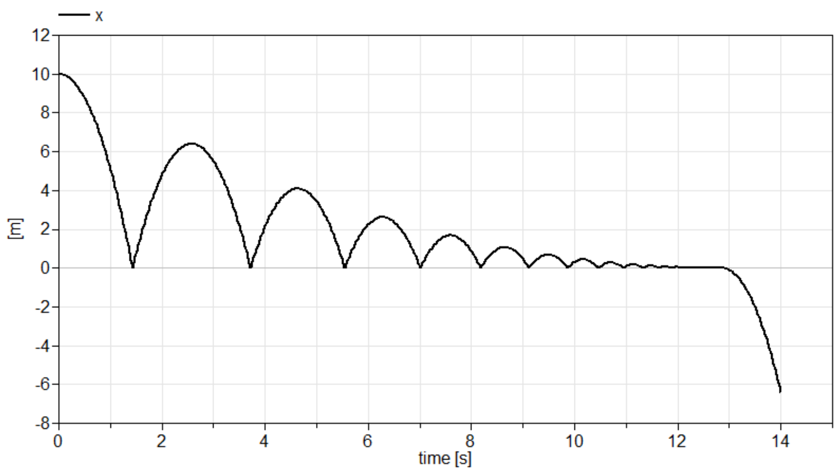


Figure 8.5: Results of simulating the Modelica Code 8.3.

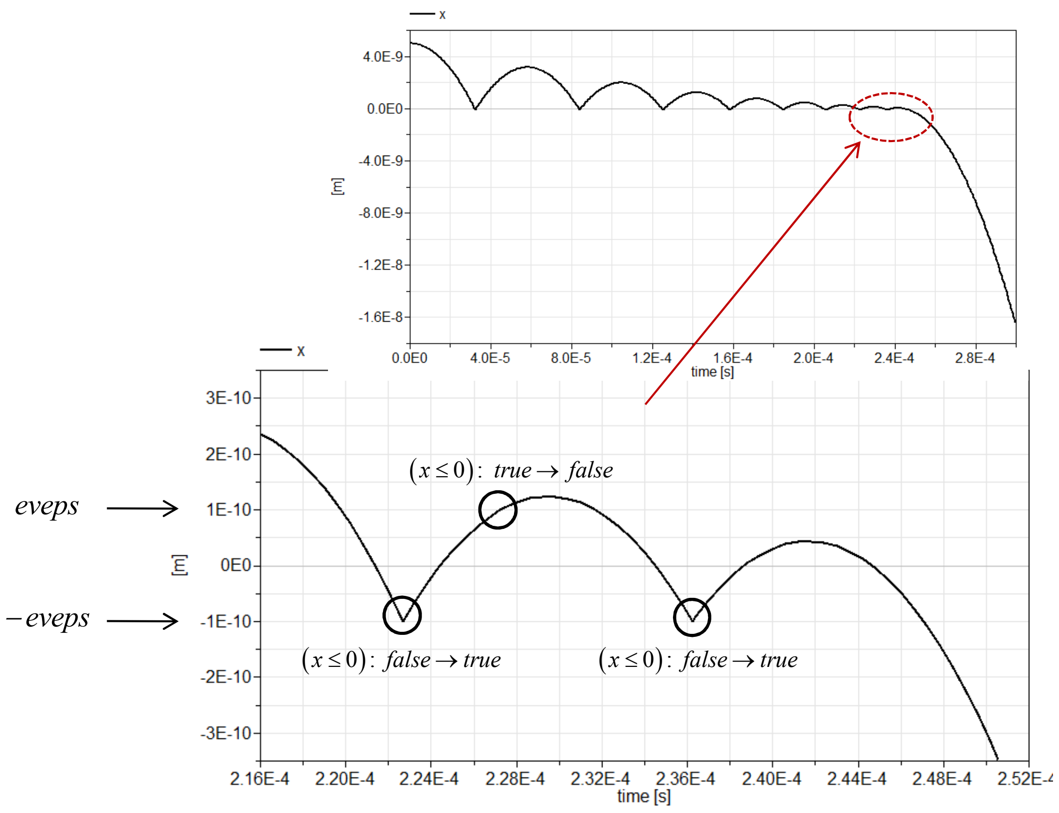


Figure 8.6: Results of simulating the Modelica Code 8.3, but with an initial height of $5E-9$ m.

8.4 Determination of the event instant

Depending on its trigger condition, events can be classified into time events and state events.

Likewise, **time events** can be classified into exogenous and endogenous. If the trigger time is specified in the model, the event is said to be **exogenous**. If this time is computed during the simulation execution as a result of the execution of a previous time event or state event, the time event is said to be **endogenous**.

During the solution of the continuous-time problem, when the next time event is scheduled within the next time step of the integration algorithm, the time step length is modified so that the evaluation time is equal to the time event. Once the event is executed, the integration algorithm is resumed, using its own method to set the size of the integration step.

The **state events** are triggered when the system state satisfies certain conditions. The trigger time of state events is not known in advanced and must be calculated during the simulation (event iteration).

To illustrate the iterative method employed to calculate the event trigger time, let's consider a **two-branch equation**, so that each branch is valid only in a certain domain of the state space. State events indicate the end of the validity domain of a branch and the start of the validity domain of the other. The modeling environment automatically define **crossing functions** for detecting state events. These functions indicate the crossing of the trajectory in the space state from one domain to the other one. When a state event is detected, the integration is halted, and the **event iteration** (iterative algorithm to determine the time instant in which the event is triggered) is started. This calculation implies the evaluation of the equation. To this end, the "old" branch is used, **extending the "old" branch beyond its validity domain**. Once the event trigger time is determined, the "old" branch of the equation is switched to the "new" branch. The **restart problem** is solved using the "new" branch of the equation. The integration algorithm is resumed, starting at the event instant, using the "new" branch of the function.

The model developer needs to take into account that the event detection procedure requires evaluating equation branches beyond their definition domain. If this is not possible, a runtime numerical error will be generated when the trajectory in the state space crosses the definition domains of the equation branches.

The following example illustrates this problem. Suppose that the mass flow rate of liquid (F) through a valve is related with the pressure drop (Δp) by means of the constitutive relationship of the valve:

$$F = \text{sgn}\,(\Delta p) \cdot K \cdot \sqrt{\Delta p} \tag{8.12}$$

where the sign function returns one if its argument is positive or zero, and minus one if negative. K is a constant that depends on the valve geometry, the friction factor and the liquid density.

If the constitutive relationship of the valve is described as follows,

$$F = \begin{array}{ll} \textbf{if} & \Delta p > 0 \\ \textbf{then} & K \cdot (\Delta p)^{0.5} \\ \textbf{else} & -K \cdot (-\Delta p)^{0.5} \end{array} \tag{8.13}$$

numerical error will be produced at runtime, when the sign of Δp changes. The reason is that the branch switching event is detected (evaluating and checking the crossing functions) after every algebraic variable has been evaluated. That is, after the square root of a real negative number has been attempted to be calculated, producing the corresponding numerical error.

The numerical error would be avoided if the simulation algorithm checks, before evaluating the two-branch equation, which of the two branches to use, which depends on the value of the condition in that time instant. The Modelica language provides an operator, named **noEvent()**, to indicate this way to proceed with a particular equation.

For example, writing in Modelica

```
F = if noEvent(difPresion > 0)
    then K*difPresion^0.5 else -K*(-difPresion)^0.5;
```

the modeling environment will check, before evaluating the right-hand side expression of the equation, which of the two branches to use.

Additionally, the noEvent() operator indicates to the modeling environment that, in case of a change of branch, it must not iterate to find the precise time instant in which the event was triggered. This way of evaluating variable structure equations, avoiding event iteration, is called performing a **textual handling** of the equation, as opposed to the **event-based handling**.

Textual handling of two-branch equations implies to integrate across the branch switching. If there exists discontinuity between the branches, the integration algorithm may fail. Integration algorithms are designed on the assumption that the function to integrate and its derivatives are continuous.

To avoid numerical errors performing event-based handling of the equations, Modelica provides some built-in functions with their branches extended beyond their domain of validity. An example is the square root function, `sqrt()`. In this way, there is no numerical error if the constitutive relationship of the valve is expressed as follows:

```
F = if difPresion > 0 then
    K*sqrt(difPresion) else -K*sqrt(-difPresion);
```

8.5 Chattering

Simulating hybrid models arise theoretical and computational problems that there not exist in continuous-time model simulation. One of these problems is the chattering. A simulation exhibits **chattering** if the number of state events executed during the simulation is large in comparison with the number of integration steps.

Every time an state event is detected, the trigger time is calculated and the restart problem is solved. Therefore, chattering significantly slows down the simulation. In some cases, the noEvent() operator allows to avoid chattering. However, this is not always the case, and the only way to avoid chattering is to modify the modeling hypotheses.

To illustrate this idea, let's consider again the model of the bouncing ball shown in Modelica Code 8.3, and let's modify it to avoid the ball to fall below the floor level. To this end, the ball acceleration is modeled making different assumptions. Now, we consider that while the ball is touching floor or below the floor level, the ball has zero acceleration. The model is shown in Modelica Code 8.4.

The result of simulating Modelica Code 8.4 during 14 s is shown in Figure 8.7. The ball does not fall below the floor level. However, when the ball energy becomes very small, which happens approximately at $t = 12$ s, the execution of the simulation slows down significantly (see the part of the plot that is encircled in the figure).

Once the simulation is finished, Dymola writes to the log window a message stating that the simulation execution has taken 3.22 seconds of CPU time, and the

```
model BouncingBall2
  Modelica.SIunits.Distance x(start=10, fixed=true);
  Modelica.SIunits.Velocity v(start=0, fixed=true);;
  parameter Real c = 0.8 "Coeff. elastic bouncing";
  parameter Modelica.SIunits.Acceleration g = 9.8;
equation
  der(v) = if (x <= 0) then 0 else -g;
  der(x) = v;
  when x <= 0 then
    reinit(v, -c*v);
  end when;
end BouncingBall2;
```

Modelica Code 8.4: Model with chattering of the vertical fall and bouncing of a ball.

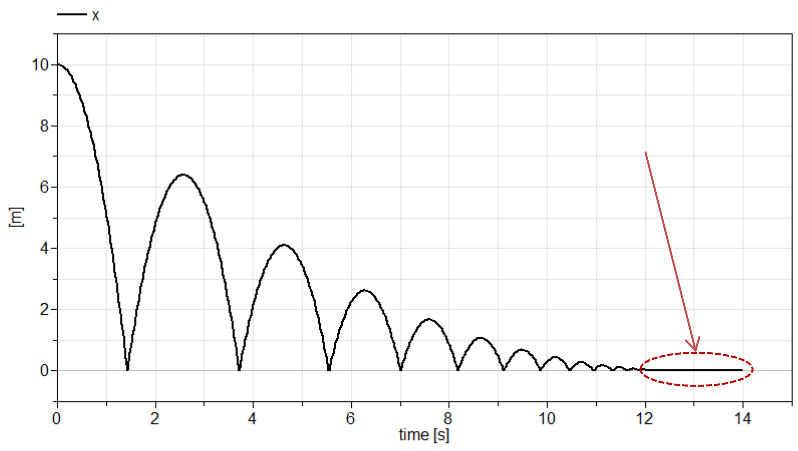


Figure 8.7: Result of simulating Modelica Code 8.4.

number of state events has been 141128. As the number of state events is very high, Dymola indicates that the model may be exhibiting chattering. Part of the message generated by Dymola is reproduced below.

```
Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dynasim))
Integration terminated successfully at T = 14
  WARNING: You have many state events. It might be due to chattering.
  Enable logging of event in Simulation/Setup/Debug/Events during simulation
   CPU-time for integration      : 3.22 seconds
   CPU-time for one GRID interval: 6.44 milli-seconds
   Number of result points       : 282757
   Number of GRID   points       : 501
   Number of (successful) steps  : 209012
   Number of F-evaluations       : 370896
   Number of H-evaluations       : 1089681
   Number of Jacobian-evaluations: 161884
   Number of (model) time events : 0
   Number of (U) time events     : 0
   Number of state    events     : 141128
   Number of step     events     : 0
   Minimum integration stepsize  : 1.75e-011
   Maximum integration stepsize  : 1.65
   Maximum integration order     : 3
Calling terminal section
... "dsfinal.txt" creating (final states)
```

An important contribution to the time spent in executing the simulation corresponds to the time employed in saving to file the simulation results. This is specially important when the model exhibits chattering. The size of this file can also become a problem for complex models.

Dymola allows the user to set the types of variables to store in file and also at which time instants. This is indicated in the *Output* tab of the *Simulation* > *Setup* window. The usual selection is to store the result at equidistant time instants and also at event instants. The time interval between these equidistant time instants is called **communication interval** (*Output interval*). The size of the communication interval or the number of communication interval is set in the *General* tab of the *Simulation* > *Setup* window.

To analyze why chattering is produced in this model, we will repeat the simulation selecting a different set of initial conditions and stop time. We consider that the ball is initially at rest with height $5E-9$ m, and the stop time is 0.0006 s. The result is shown in Figure 8.8. Note that the ball bounces periodically from $4E-4$ s, and a bounce (i.e., state event) takes place every approximately $2E-5$ s.

The ball shows a periodic behavior in the stationary, bouncing indefinitely. This behavior is not described in the model. It is a numerical artifact due to the mechanism for event detection used by Dymola. Note that while the ball goes up from $-eveps$ to $+eveps$, the event condition $x \leq 0$ is *true*. Thus, the gravity is not acting on the ball during these time periods. The energy required for the ball to ascend from $-eveps$ to $+eveps$ is given to the ball. When this energy provided “for free” to the ball (as a result of the event detection procedure) becomes equal to energy lost in the bouncing, the model oscillates indefinitely.

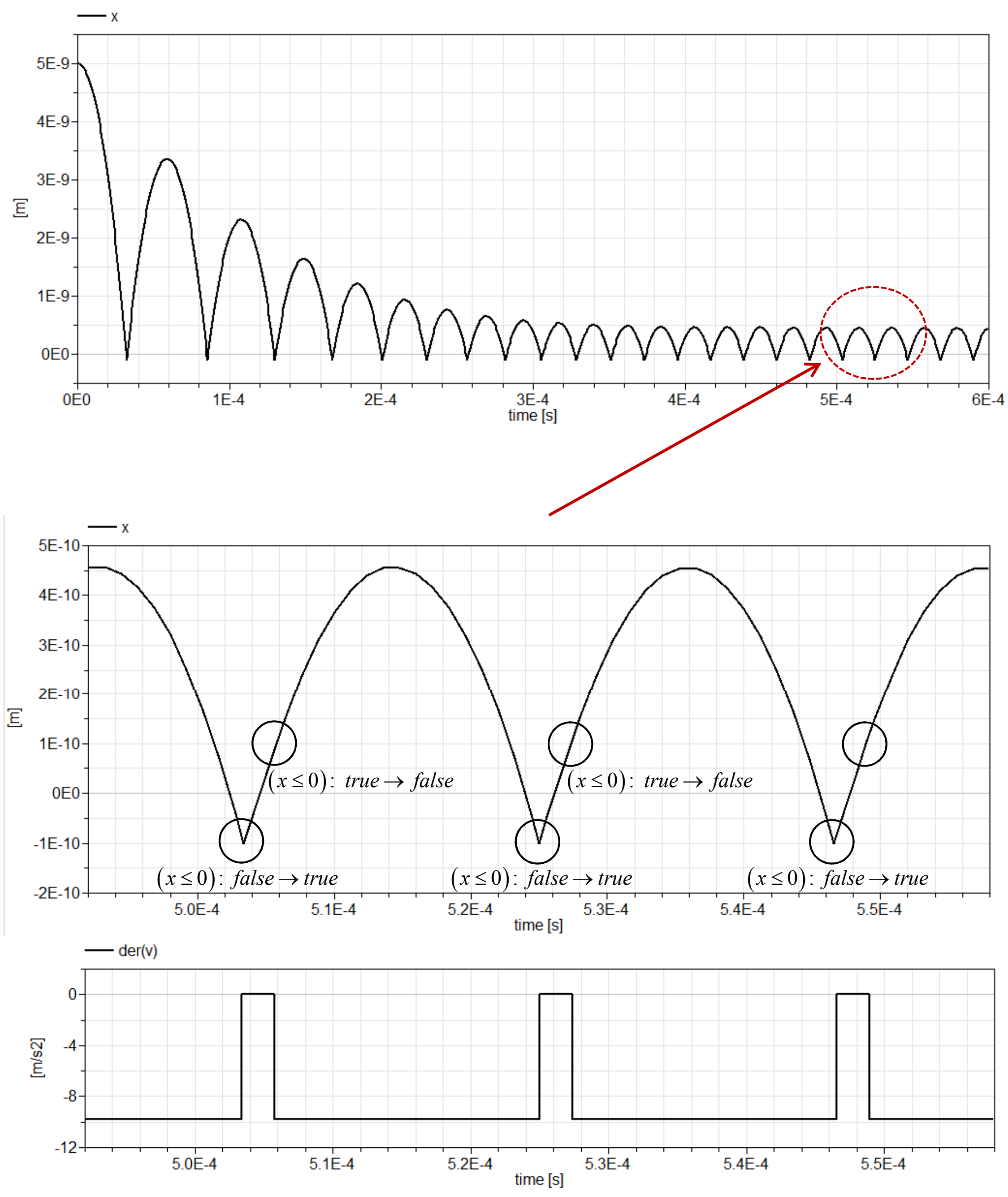


Figure 8.8: The ball reaches a stationary periodic behavior, which is a numerical artifact originated by the event detection procedure.

8.6 Further reading

The discussion on the treatment of simultaneous events was extracted from (Andersson 1994). Event detection and handling are discussed in (Cellier 1979), and in Chapter 9 of (Cellier & Kofman 2006). The use of this thesis report and this book was recommended in previous lessons. Another excellent reference on hybrid system simulation is (Barton 1992).

The event detection in Dymola, employing two crossing functions, is described in (Elmqvist et al. 1993). A discussion on the branch switching mechanism employed for simulating if-sentences, and the numerical errors originated by the event detection mechanism, can be found in (Elmqvist et al. 1994).

Lesson 9

Hybrid modeling practice

9.1 Introduction
9.2 Ideal electric switch
9.3 Ideal diode
9.4 Two-tank and valve system
9.5 Bouncing ball
9.6 Dry fiction
9.7 Heat conduction in a wall
9.8 Further reading

Learning objectives

After studying the lesson, students should be able to:

– Develop multi-mode models in Modelica.

– Design, develop and use hybrid model libraries in Modelica.

9.1 Introduction

Hybrid models are those that combine continuous-time behavior with events. An **event** is a set of actions that are triggered when a certain condition is satisfied. Therefore, the definition of an event consists of specifying the logic condition that triggers it, and the actions to be performed.

The execution of the actions associated to the event does not consume simulated time. For this reason, those variables that change due to the event have two values at the time in which the event is triggered: the value before the event execution and the value after the event execution.

Modelica provides the function **pre**() to distinguish between the variable value before the event, **pre**(*variable*), and the new value of the variable after the event execution, which is referred using the name of the variable. For example, if the instantaneous equation that describes the action associated to an event is:

$$x = 2 \cdot \mathbf{pre}(x) \tag{9.1}$$

then the value of the x variable is doubled each time this event is executed.

The actions that Modelica allows to perform in an event are basically of the following three types:

- **Change in the model structure**. An event can generate a change in the mathematical structure of the model. That is, a change in the equations that describe the model behavior.
- **Update the value of discrete-time variables**. The action associated to an event can be to modify the value of one or more discrete-time variables. The value of a discrete-time variable is constant between two consecutive events, changing only at event instants.
- **Reinitialization of continuous-time state variables**. Other action associated to an event can be to change the value of a continuous-time variable. For this change to take effect, the variable whose value is reinitialized in the event action has to be a state variable.

If sentences and if clauses are employed to describe the first type of action. For the last two, the when clause is employed. This is explained in detail below.

9.1.1 If sentence and clause

The if-sentence and the if-clause allow to describe models with a variable structure. Both can be included in equation and algorithm sections.

As shown in the previous lessons, the **if sentence** allows to describe functions with several branches. It has basically the following syntax, when written inside an equation and algorithm section respectively:

$$\begin{aligned}
\mathit{expr}_1 = &\quad \textbf{if}\ \mathit{cond}\ \ \textbf{then}\ \ \mathit{expr}_2\ \ \textbf{else}\ \ \mathit{expr}_3; \\
\mathit{var} := &\quad \textbf{if}\ \mathit{cond}\ \ \textbf{then}\ \ \mathit{expr}_1\ \ \textbf{else}\ \ \mathit{expr}_2;
\end{aligned} \tag{9.2}$$

where *var* is a variable, *cond* is a Boolean expression, and *expr* is an expression. Else branches can be replaced by elseif-then-else. For example:

$$\begin{aligned}
\mathit{expr}_1 = &\quad \textbf{if}\ \mathit{cond}_1 \quad\ \ \textbf{then}\ \ \mathit{expr}_2 \\
&\quad \textbf{elseif}\ \ \mathit{cond}_2\ \ \textbf{then}\ \ \mathit{expr}_3\ \ \textbf{else}\ \ \mathit{expr}_4; \\
\mathit{var} := &\quad \textbf{if}\ \mathit{cond}_1 \quad\ \ \textbf{then}\ \ \mathit{expr}_1 \\
&\quad \textbf{elseif}\ \ \mathit{cond}_2\ \ \textbf{then}\ \ \mathit{expr}_2\ \ \textbf{else}\ \ \mathit{expr}_3;
\end{aligned} \tag{9.3}$$

The syntax of an **if clause** written within an equation section is basically:

```
if  cond  then
      equations
else                                  (9.4)
      equations
end if;
```

If clauses written inside algorithm sections contain assignments, instead of equations. Analogously to the if-sentences, the else branches can be replaced by elseif-then-else. For example,

```
if  cond1  then
      equations
elseif  cond2  then
      equations                       (9.5)
else
      equations
end if;
```

9.1.2 Textual handling of if expressions

As explained in previous lessons, Modelica modeling environments perform an event-based handling of if expressions. This is, when an event is detected, the numerical integration of the continuous-time problem is halted, the trigger time is calculated by applying iterative algorithms (event iteration), and the restart problem is solved. The old branch of the if expression is used in the event iteration, and the new branch in solving the restart problem.

However, in some cases the branch switching does not introduce any discontinuities in the value of the expression and its derivatives, or the effect of these is small. In these cases, it is possible to integrate over the branch switching point. As this procedure avoids to perform the event iteration (calculate precisely the time in which the branch switching is produced), the simulation CPU-time is reduced. Performing such a textual treatment of the if expression can result in a significant reduction in the simulation time of models with a large number of events.

Using the **noEvent()** function in the logical condition of an if expression indicates that the if expression has to be handled textually (in opposition to the by-default event-based handling). For example, given the following sentences, written inside equation or algorithm sections respectively

$$\begin{aligned} &\mathit{expr}_1 = \textbf{if noEvent}(\mathit{cond})\ \textbf{then}\ \mathit{expr}_2\ \textbf{else}\ \mathit{expr}_3; \\ &\mathit{var} := \textbf{if noEvent}(\mathit{cond})\ \textbf{then}\ \mathit{expr}_1\ \textbf{else}\ \mathit{expr}_2; \end{aligned} \tag{9.6}$$

the modeling environment evaluates in first place the Boolean condition, to choose which one of the two branches of the if expression must use, and then computes the corresponding variable using the chosen branch.

As discussed in Section 8.4, the noEvent() function also allows to avoid runtime numerical errors when the branches cannot be extended beyond its validity range. For example, the if sentence

$$y = \ \textbf{if}\ u >= 0\ \textbf{then}\ u\hat{}0.5\ \textbf{else}\ (-u)\hat{}0.5; \tag{9.7}$$

can generate an execution error of the type “attempt to compute the square root of a negative number”. This error is avoided using noEvent():

$$y = \ \textbf{if noEvent}(u >= 0)\ \textbf{then}\ u\hat{}0.5\ \textbf{else}\ (-u)\hat{}0.5; \tag{9.8}$$

9.1.3 When clause

The syntax of the when clause is basically the following:

$$\begin{array}{l} \textbf{when } \textit{cond} \textbf{ then} \\ \qquad \text{instantaneous equations} \\ \textbf{end when}; \end{array} \tag{9.9}$$

where *cond* (the clause's trigger condition) is a Boolean expression. In this case, the instantaneous equations included in the body of the when clause are executed only at the time instant in which the Boolean expression value changes from *false* to *true*.

The trigger condition of a when clause can also be a vector of Boolean expressions, as shown below. In this case, the when clause is triggered each time any of the vector components changes its value from *false* to *true*.

$$\begin{array}{l} \textbf{when } \{cond_1, \dots, cond_n\} \textbf{ then} \\ \qquad \text{instantaneous equations} \\ \textbf{end when}; \end{array} \tag{9.10}$$

To illustrate the difference between these two ways of specifying the trigger condition, consider the two when clauses shown below:

```
when u1>0 or u2>0 then
   b1 = not pre(b1);
end when;
```

```
when { u1>0 , u2>0 } then
   b2 = not pre(b2);
end when;
```

The clause written on the left-hand side is triggered when the value of the Boolean expression `(u1>0 or u2>0)` changes from *false* to *true*. The clause on the right-hand side is triggered when the value of `u1>0` changes from *false* to *true*, and also when the value of `u2>0` changes from *false* to *true*. See Figure 9.1.

The **instantaneous equations** can be of two types:

– **Difference equations** describing how the new values of discrete-time variables are evaluated. Depending on whether the when clause is included inside an equation or algorithm section, the instantaneous equation has the following syntax respectively, where *var* represents the new value of the variable.

$$var = expr; \tag{9.11}$$

$$var := expr; \tag{9.12}$$

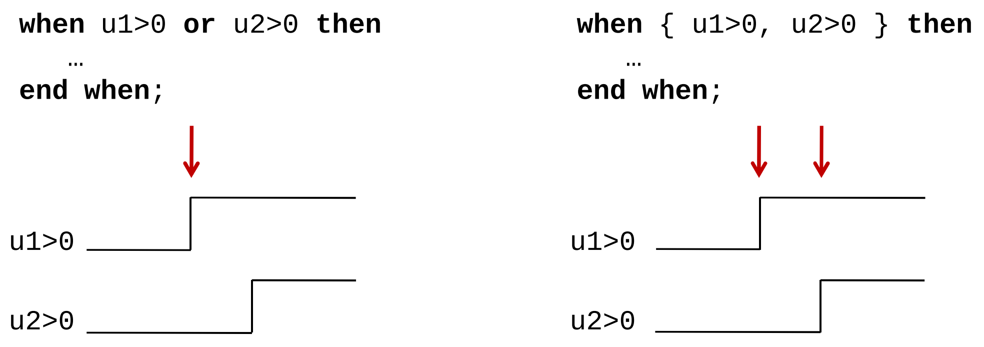


Figure 9.1: Trigger conditions expressed as a Boolean expression (left), and as an array of Boolean expressions (right). Trigger instants are indicated by arrows.

The **pre**() function allows to refer to the variable value before the event, e.g., **pre**(*variable*). Observe that the difference equations have to be written in explicit form. This is, with the variable to evaluate written on the left-hand side of the "=" or ":=" symbol. The new value of the variable is the result of evaluating the expression written on the right-hand side of the difference equation.

– **reinit sentences**, employed to change abruptly the value of continuous-time state variables. The function has two arguments. The first one is the continuous-time state variable whose value is going to be changed. The second one is the expression used to evaluate the new value of the state variable. If the first argument of the reinit function is not an state variable, then the function call has not any effect. The function is invoked as follows:

$$\textbf{reinit}(var, expr); \tag{9.13}$$

The **single-assignment rule** applies within the when-clause body depending on whether the when clause is written inside an equation section or an algorithm section. This is explained below.

– In the case of a when clause written within an **equation section**, the order in which the model developer writes the instantaneous equations is irrelevant. The modeling environment sorts automatically the instantaneous equations. For this to be possible, the single-assignment rule must be fulfilled, of which it is being guaranteed that each variable is evaluated from a single equation.

– The assignments included inside an **algorithm section** are considered an indivisible set, and the modeling environment does not manipulate them or change their order. Therefore, the single assignment rule doesn't apply inside an algorithm section: several assignments to the same variable can be written within an algorithm section. For example, it is possible to write:

```
algorithm
   when h1>3 then
      closed := true;
   end when;
   when h2>1 then
      closed := false;
   end when;
```

If these two events are triggered simultaneously, then they are executed in order and, therefore, the new value of the `closed` variable would be *false*.

Nevertheless, the algorithm sections have to be sorted together with the rest of the model. For this reason, a variable calculated in an algorithm section cannot be calculated in any other algorithm section, or from an equation.

The *sample*(), *initial*() and *terminal*() built-in functions can be used in the **Boolean condition** of the when clause as described below.

– The **sample()** function triggers periodically the when clause, starting at a determined initial instant. For example, the clause written below is triggered at the following time instants: $t_0 + n \cdot T$, with $n = 0, 1, 2, \dots$

```
when sample(t0, T) then
   x = a*pre(x) + b*u;
   y = c*pre(x) + d*u;
end when;
```

– As discussed in Section 7.6.4, the **initial()** function triggers the when clause at the model initialization.

– The **terminal()** function triggers the when clause when the ending condition of the simulation is satisfied. It has the following syntax:

$$\begin{array}{l} \textbf{when } \mathit{terminal}() \textbf{ then} \\ \dots \\ \textbf{end when}; \end{array} \tag{9.14}$$

The when clause can also be used to force the **simulation termination**. The ending condition in the examples discussed so far is the simulated time to reach a predefined value, which is specified in the experiment definition. However, the termination condition of some simulation studies depends not only on time, but also on some model variables.

These termination conditions, dependent on model variables, can be described by calling the **terminate()** function inside the body of a when clause. This built-in function has one argument of String type: the message written by the modeling environment to the message window when the function is executed, forcing the simulating to finish. For example:

```
   Modelica.SIunits.Temperature T           "Mixture temperature";
   parameter Real eps (unit="K/s") = 1e-3  "Small value";
equation
   when abs(der(T)) < eps then
      terminate("A steady-state has been reached");
   end when;
```

String is a predefined type of variable in Modelica. String variables store chains of characters. To work with String variables, it is useful to know that:

– The + operator concatenates Strings.

– Strings can contain end-of-line (\n) and tab (\t) characters.

– The **realString()** and **integerString()** functions allow to convert Real and Integer types into String, respectively. These functions have three input arguments, and one output argument, as shown below.

```
function realString "Convert a real to a string"
   input Real number       "The number to convert to a string";
   input Integer minimumWidth := 1   "Minimum width of result";
   input Integer precision := 6 "Number of significant digits";
   output String result;

function integerString "Convert an integer to a string"
   input Integer number    "The number to convert to a string";
   input Integer minimumWidth := 1   "Minimum width of result";
   input Integer precision := 1     "Minimum number of digits";
   output String result;
```

Another application of the when clause is to **write to the message window**. This can be useful to debug models, as it allows to get information about the value of determined variables at specified simulation time instants, or when certain conditions are satisfied.

The **LogVariable()** function writes the actual value of a variable to the message window. For example, the following when clause writes the value of the x variable every 0.1 s, starting from time zero.

```
when sample(0,0.1) then
   LogVariable(x);
end when;
```

9.2 Ideal electric switch

Let's consider the model of an electric circuit shown in the Figure 9.2, that is composed of a voltage generator, a resistor and an ideal switch. The Boolean variable *open*, whose time evolution is shown on the right-hand side of the figure, is a known function of time. The Modelica description of this circuit is shown in Modelica Code 9.1.

The if sentence shown below has been used to describe the constitutive relationship of the ideal electric switch. Comparing this constitutive relationship with Eq. (7.83), it can be observed the analogy between the electrical and hydraulic domains.

$$0 = \textbf{if } open \textbf{ then } i \textbf{ else } u_D; \tag{9.15}$$

When the model is translated, Dymola shows a warning stating that the units of the if-expression branches are different: the i branch is expressed in amperes, and the u_D branch in volts. This is not an error in this model, so we can ignore the warning.

The results obtained executing the simulation during 16 s are shown in Figure 9.3. While the switch is open, the current is zero and the voltage drops across the switch. While the switch is closed, there is a flow of electrical current and the voltage drops across the resistor.

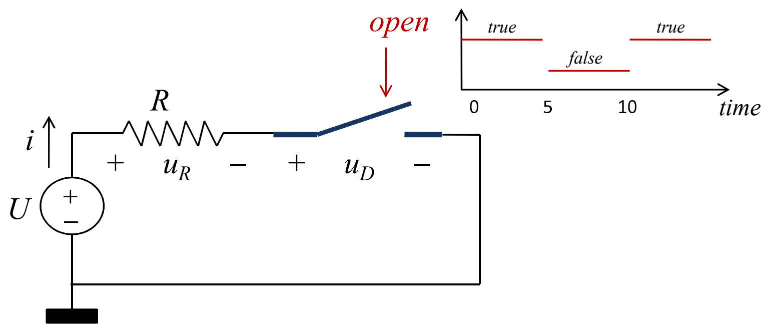


Figure 9.2: Electric circuit, and evolution of the *open* Boolean variable.

```
 model ResistConmutCircuit
  Modelica.SIunits.Voltage U, uR, uD;
  Modelica.SIunits.Current i;
  Boolean open;
  parameter Modelica.SIunits.Resistance R = 10;
  parameter Modelica.SIunits.Voltage U0 = 5;
  parameter Modelica.SIunits.AngularFrequency w = 2;
equation
  U = U0*sin(w*time);    // Generator
  uR = i*R;    // Resistor
  0 = if open then i else uD;    // Switch
  open = time < 5 or time > 10;   // Control of switch
  U = uR + uD;
end ResistConmutCircuit;
```

Modelica Code 9.1: Model of the electric circuit shown in Figure 9.2.

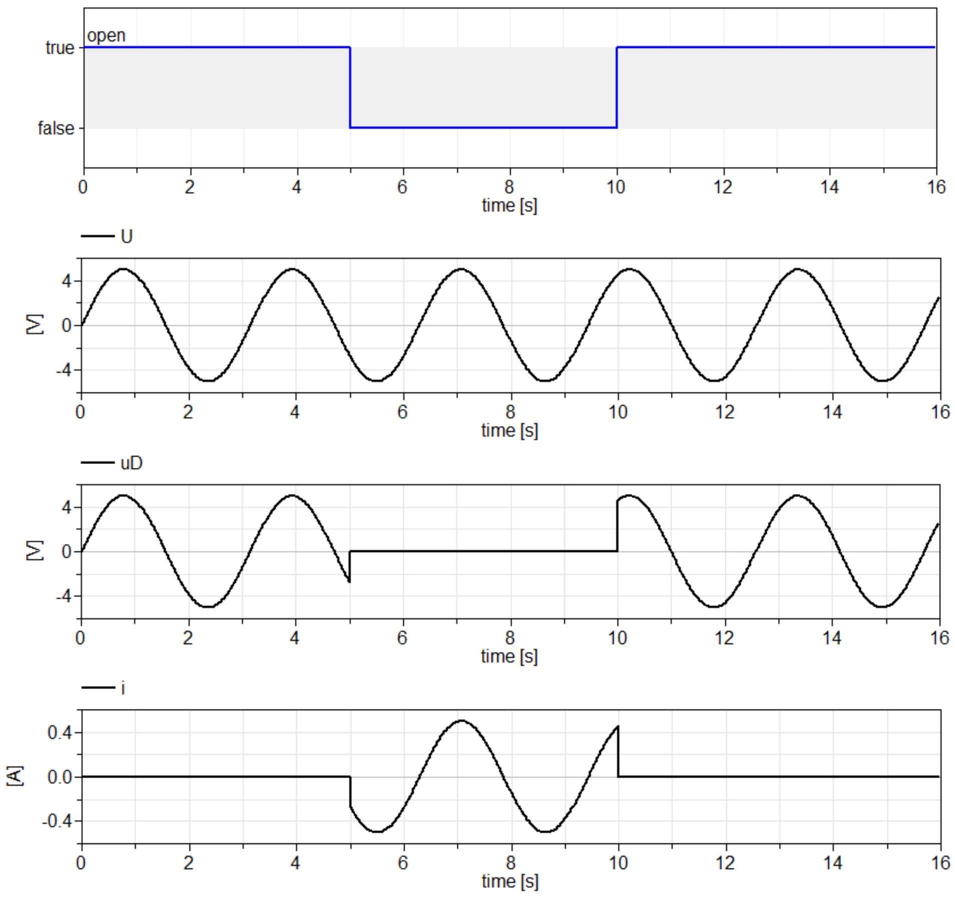


Figure 9.3: Simulation of Modelica Code 9.1.

9.3 Ideal diode

The model of the switch can be modified to describe an ideal diode. In the diode, the opening condition depends on internal variables of the device, instead of being determined by a particular function of time.

An ideal diode can be in two modes, *conduction* and *cutoff*, that are analogous to the closed and open modes of an ideal switch. The behavior of the ideal diode is modeled in the following way (see Figure 9.4):

– While in the **conduction** phase, the voltage drop across the diode terminals is zero, and the current flowing through it must have a positive sign. When this last condition is not satisfied, the diode changes to the *cutoff* phase.

– While in the **cutoff** phase, the current flowing through the diode is zero, and the voltage drop across the diode terminals must be less or equal to zero. When this condition on the voltage drop is not satisfied, the diode changes to the *conduction* phase.

The constitutive relationship of the ideal diode can be described in Modelica by means of the two Eqs. (9.16). Note that the Boolean variable *corte* is calculated from the second equation, evaluating a logical expression that depends on two continuous-time variables. While *corte* equals *true*, the diode is in the *cutoff* mode; and while *false*, in the *conduction* mode.

$$\begin{aligned} 0 &= \textbf{if } corte \textbf{ then } i \textbf{ else } u_D; \\ corte &= i \leq 0 \textbf{ and not } u_D > 0; \end{aligned} \tag{9.16}$$

To illustrate the application of the previous model, let's consider the rectifier circuit shown in Figure 9.5, whose model is described by the following equations.

$$u = U_0 \cdot \sin(w \cdot t) \tag{9.17}$$

$$u - u_1 = i_1 \cdot R_1 \tag{9.18}$$

$$0 = \textbf{if } corte \textbf{ then } i_1 \textbf{ else } u_1 - u_2 \tag{9.19}$$

$$corte = i_1 \leq 0 \textbf{ and not } u_1 - u_2 > 0 \tag{9.20}$$

$$u_2 = i_2 \cdot R_2 \tag{9.21}$$

$$C \cdot \frac{du_2}{dt} = i_C \tag{9.22}$$

$$i_1 = i_2 + i_C \tag{9.23}$$

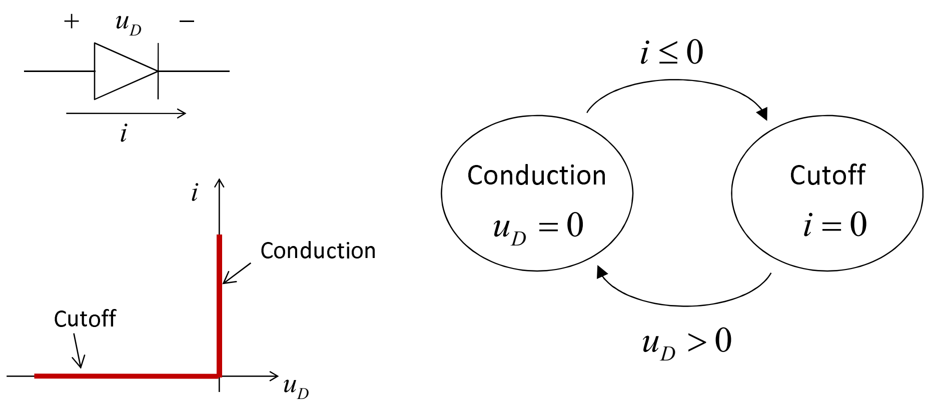


Figure 9.4: I-V characteristic curve of an ideal diode and its two-phase model.

The results shown in Figure 9.6 are obtained simulating Modelica Code 9.2 during 0.1 s. The circuit behavior is as follows.

– While the diode is in the **cutoff** mode ($corte = true$), the current through the diode (i_1) is zero. Therefore, there is no voltage drop in the R_1 resistor, and u_1 is equal to u.

– While the diode is in the **conduction** mode ($corte = false$), there is no voltage drop across its pins ($u_1 = u_2$). Part of the generator voltage drops in the R_1 resistor, and the rest drops in the parallel of R_2 and C.

In analogy to the problem discussed for the ideal switch, the ideal diode model produces an error if the mode change implies a change in the number of DoF of the complete model. For example, the ideal diode model cannot be used to model the circuit shown in Figure 9.7. The reason is as follows.

– The circuit model has one DoF while the diode is in the **cutoff** mode. The constitutive relationship of the diode forces that no current flows through it. Thus, the current provided by the generator, that flows completely through R_1, is distributed between R_2 and C. The voltage drop across the capacitor can be selected as state variable.

– However, while the diode is in the **conduction** mode, the constitutive relationship of the diode forces the voltage drop across its terminals to be zero. As a consequence, the voltage drop across the condenser is computed from the diode constitutive relationship and this variable is not a state variable. The model has zero DoF.

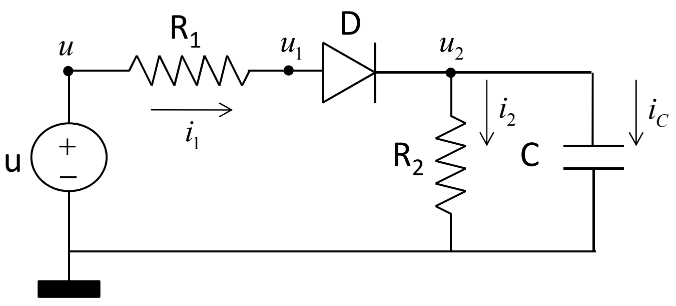


Figure 9.5: Rectifier circuit.

```
model idealDiodeCircuit
  import SI = Modelica.SIunits;
  SI.Current i_1, i_2, i_C;
  SI.Voltage u, u_1, u_2(start=0, fixed=true);
  Boolean corte "Diode mode";
  parameter SI.Voltage U0=5;
  parameter SI.Frequency frec=50;
  parameter SI.AngularFrequency w=2*Modelica.Constants.pi*frec;
  parameter SI.Resistance R1=10, R2=50;
  parameter SI.Capacitance C=1e-3;
equation
  u = U0*sin(w*time);
  u - u_1 = i_1*R1;
  0 = if corte then i_1 else u_1 - u_2;
  corte = i_1 <= 0 and not u_1 - u_2 > 0;
  u_2 = i_2*R2;
  C*der(u_2) = i_C;
  i_1 = i_2 + i_C;
end idealDiodeCircuit;
```

Modelica Code 9.2: Rectifier circuit shown in Figure 9.5 with ideal diode.

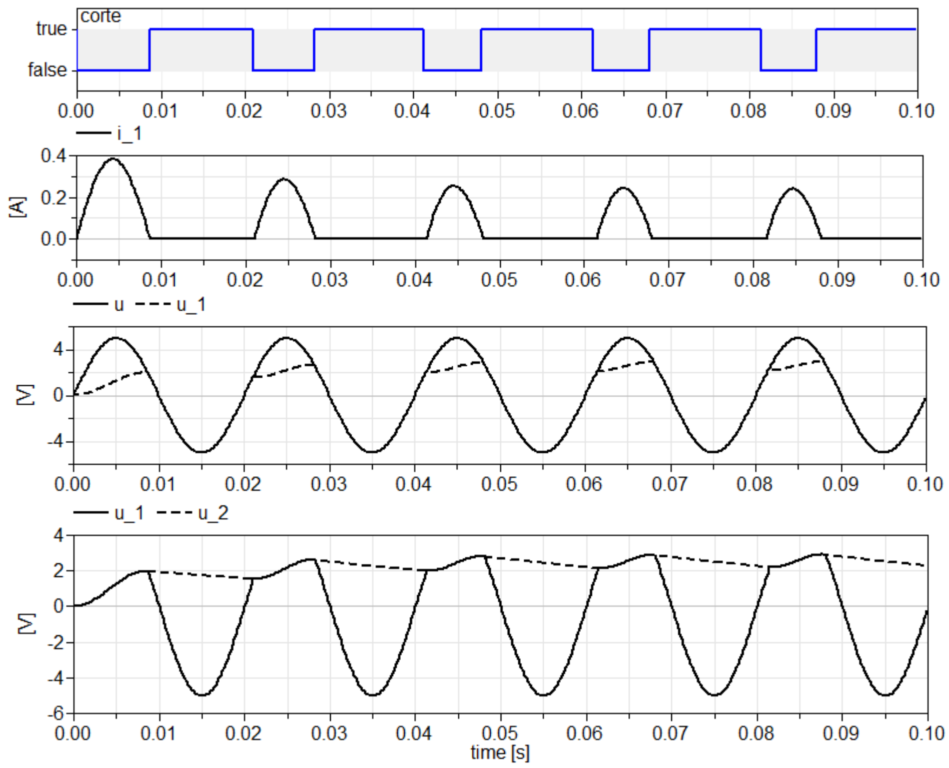


Figure 9.6: Results obtained simulating Modelica Code 9.2.

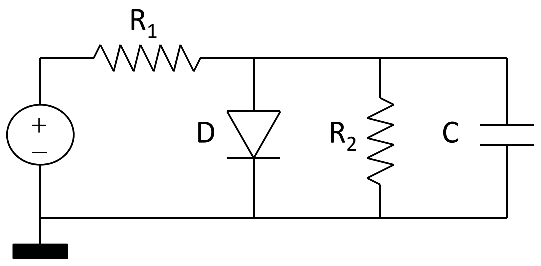


Figure 9.7: The ideal diode model cannot be employed in modeling this circuit.

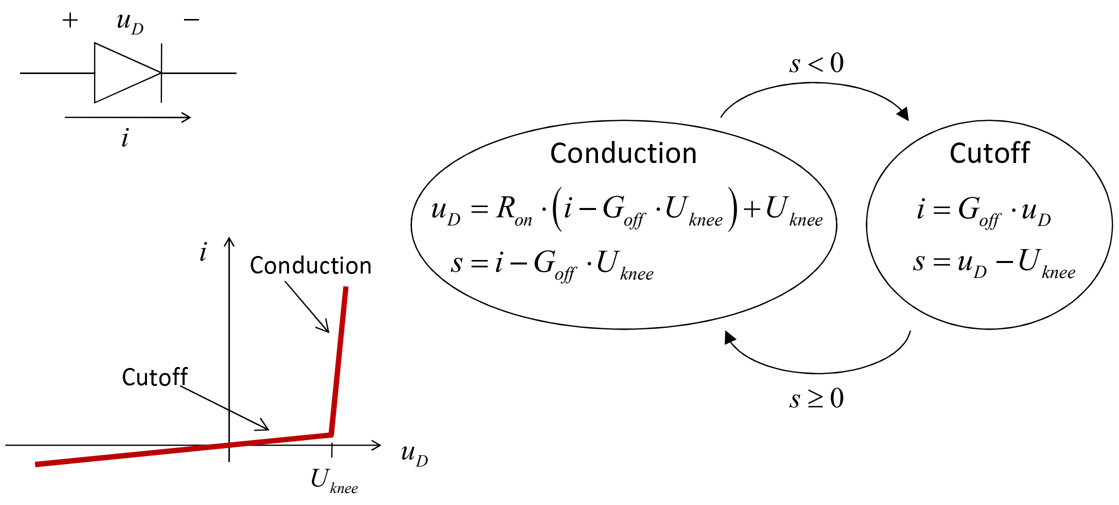


Figure 9.8: Diode model included in the Modelica Standard Library 3.2.1.

To avoid that a change in the diode mode generates a change in the number of DoF of the complete circuit model, the diode model included in the Modelica Standard Library (MSL) includes a little resistance (R_{on}) while in conduction, and a little conductance (G_{off}) while in cutoff. The constitutive relationship of this **resistive diode**, in the version 3.2.1 of the MSL distributed with Dymola 2015, is the following (see Figure 9.8):

$$\begin{aligned} corte &= s < 0 \\ u_D &= (s \cdot unitCurrent) \cdot (\textbf{if}\ \ corte\ \textbf{then}\ 1\ \textbf{else}\ R_{on}) + U_{knee} \\ i &= (s \cdot unitVoltage) \cdot (\textbf{if}\ \ corte\ \textbf{then}\ G_{off}\ \textbf{else}\ 1) + G_{off} \cdot U_{knee} \end{aligned} \tag{9.24}$$

where the U_{knee} parameter is the threshold voltage of the diode, i.e., the value of u_D for which the slope of the I-V characteristic changes (see Figure 9.8). By default, U_{knee} is zero, the R_{on} resistance is 10^{-5} ohm, and the G_{off} conductance is 10^{-5} ohm^{-1}. The values of these three parameters can be modified when the component is instantiated, and when the experiment is defined.

The s variable, that determines the diode operating point, is declared as a dimensionless variable (`unit="1"`) of Real type. While the diode is in cutoff, s is equal to $u_D - U_{knee}$. While the diode is in conduction, s is equal to $i - G_{off} \cdot U_{knee}$. If $s = 0$, the diode operates in the threshold voltage (U_{knee}). While $s < 0$, the operation point is below the threshold voltage and the diode is in the *cutoff* mode. While $s > 0$, the operation point is above the threshold voltage, and the diode is in the *conduction* mode.

The $unitCurrent$ and $unitVoltage$ constants have been declared to avoid inconsistency in the units of Eq. (9.24).

```
constant Modelica.SIunits.Voltage unitVoltage = 1;
constant Modelica.SIunits.Current unitCurrent = 1;
```

9.4 Two-tank and valve system

Let's consider the system described in Section 7.4.3, which is composed of two tanks connected by a valve. For readers' convenience, the system diagram is shown again in Figure 9.9. The system is modeled as shown in Modelica Code 9.3. If sentences are used to describe the mass flow rate and the temperature of the liquid that flows through the valve. A when sentence is used to describe the change of the valve opening, which occurs when time becomes greater than $t_0 = 50$ s. The simulation result is shown in Figure 9.10.

Observe that while the value of the `theta` variable is zero, the flow between the two tanks is zero. Thus, the mass and temperature remain constant in each tank. When the valve opening `theta` is set to the value 0.5 (at $t = 50$ s), the mass flow between the two tanks becomes instantaneously different from zero. Since the liquid level is at that moment higher in the second tank, the liquid flows from the second tank to the first one. The liquid that flows through the valve is at the same temperature as the liquid stored in the upstream tank. Therefore, the flowing liquid is at the temperature of the second tank. The liquid temperature in the second tank is higher than the liquid temperature in the first tank, and consequently the liquid temperature in the first tank increases.

As the liquid level difference tends to zero, the flow through the valve tends to zero. When the levels (and thus the pressures) become equal, the flow becomes zero. As no more liquid enters the first tank, the temperature of the stored liquid remains constant from that instant. The liquid in the second tank maintains the same temperature during the entire simulation.

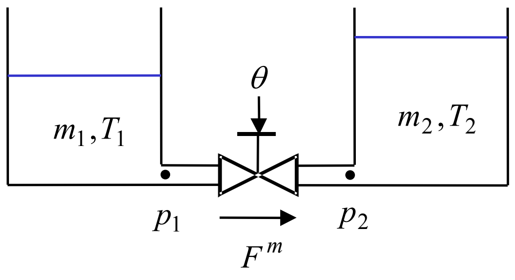


Figure 9.9: Two-tank and valve system described in Section 7.4.3.

```
 model TanksValve
   import SI = Modelica.SIunits;
   constant SI.Acceleration g=9.81 "Gravitational acceleration";
   parameter SI.Area S1 = 1, S2 = 2;
   parameter Real Kv = 0.2;
   parameter Real theta0(unit="1") = 0.5;
   parameter SI.Time t0 = 50;
   parameter SI.Density rho = 1000;
   SI.Mass m1(start=1000,fixed=true), m2(start=4000,fixed=true);
   SI.Temperature T1(start=300,fixed=true), T2(start=350,fixed=true), Tf;
   SI.Height h1, h2;
   SI.Pressure p1, p2;
   SI.MassFlowRate Fm;
   Real theta( start=0, fixed=true, unit="1");
equation
   // Tank 1
   der(m1) = -Fm;
   m1 * der(T1) = -Fm * (Tf-T1);
   p1 = m1 * g / S1;
   m1 = rho * h1 * S1;
   // Tank 2
   der(m2) = Fm;
   m2 * der(T2) = Fm * (Tf-T2);
   p2 = m2 * g / S2;
   m2 = rho * h2 * S2;
   // Valve
   Tf = if p1 > p2 then T1 else T2;
   Fm = if p1 > p2 then  Kv*theta*sqrt(abs(p1-p2))
                   else -Kv*theta*sqrt(abs(p2-p1));
   when time > t0 then
      theta = theta0;
   end when;
end TanksValve;
```

Modelica Code 9.3: Model of the two-tank and valve system shown in Figure 9.9.

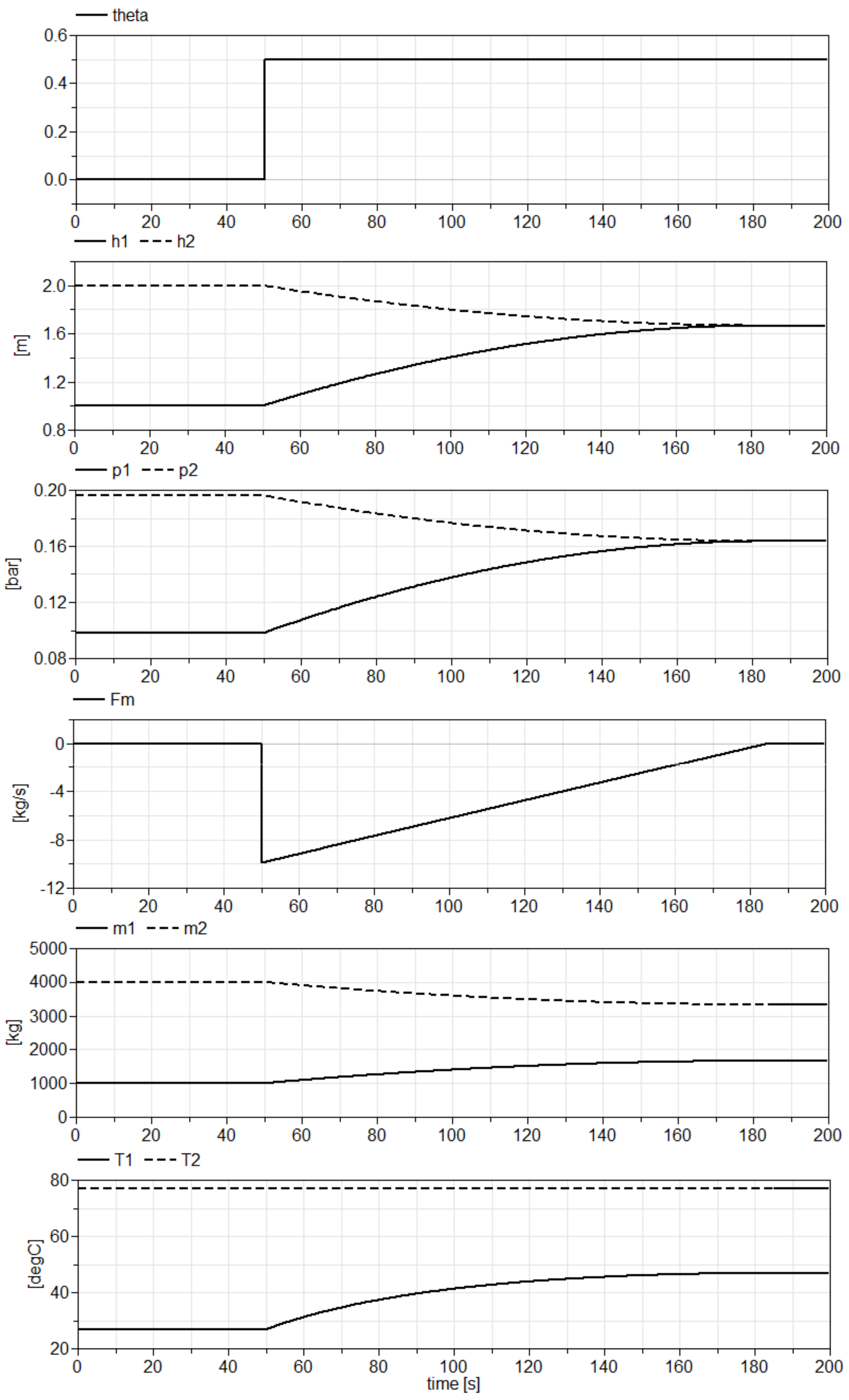


Figure 9.10: Simulation of Modelica Code 9.3 during 200 s.

9.5 Bouncing ball

The vertical movement of a ball falling under the gravity action and bouncing on the floor was modeled in Section 7.4.1. Two different models were proposed that, as we have seen, were not satisfactory: the Modelica Code 8.3 allows the ball to fall below the floor level, and the Modelica Code 8.4 exhibits chattering. Let's adopt a different approach in modeling the bouncing ball behavior.

The **specific energy** (energy per mass unit) of the ball is defined by Eq. (9.25). The first term in the right-hand side expression is the ball's kinetic energy, and the second term its potential energy.

$$e = \frac{1}{2} \cdot v^2 + g \cdot x \tag{9.25}$$

The ball's specific energy is computed at the beginning of the simulation, $e_{inicial}$, and after each bounce. When the specific energy after a bounce becomes less than a determined proportion of the initial one, the ball enters in a mode, named *Stoppe-dOnFloor*, in which the position and the velocity are forced to be zero. Notice that the computational load of the model while in the *StoppedOnFloor* mode is very low, and the chattering problem is avoided.

To avoid the ball to fall below the floor level, it is checked after each bounce whether the ball's kinetic energy is large enough to ascend over the $(2 \cdot eveps)$ height. Otherwise, the simulation is finished, and a message warning that the model is out of its experimental frame is written to the log window. This indicates that the model is being used in an experimental context for which the model is not valid.

The behavior of the model is shown in Figure 9.11. The point filled in black is the entry to the diagram. From this point, it is triggered an unconditional transition from the *Moving* mode, where the model describes the fall and bounce of the ball. The ball energy changes with each bounce.

If the ball goes out of its experimental frame, the simulation is finished. The simulation end is represented by two concentric circles, so that the outer circle is hollow and the inner circle is filled in black.

If, inside the experimental frame, the specific energy is lower than $coef \cdot e_{inicial}$, a transition to the *StoppedOnFloor* mode takes place, and the model stays in this mode indefinitely.

The model description is shown in Modelica Code 9.4. Notice that:

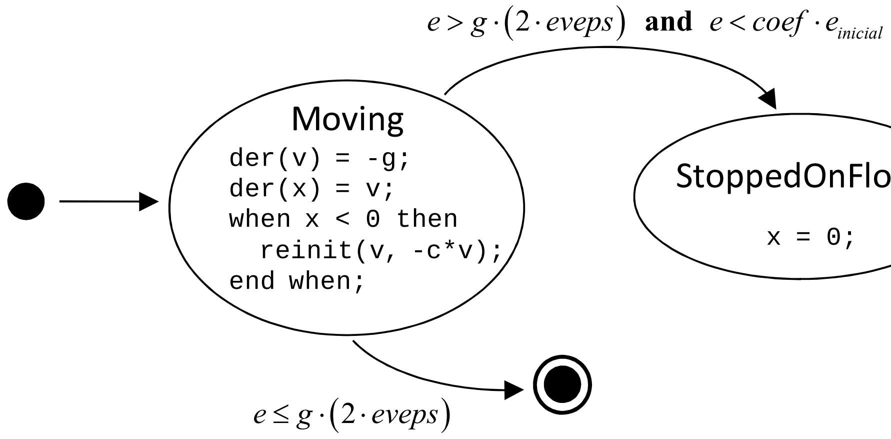

$e > g \cdot (2 \cdot eveps)$ **and** $e < coef \cdot e_{inicial}$
Moving
der(v) = -g;
der(x) = v;
when x < 0 then
reinit(v, -c*v);
end when;
StoppedOnFloor
x = 0;
$e \leq g \cdot (2 \cdot eveps)$

Figure 9.11: Model of the bouncing ball with two modes: *Moving* and *StoppedOnFloor*.

```
model BouncingBall3

  import SI = Modelica.SIunits;

  constant Real eveps(unit="1") = 1E-10;
  constant SI.Acceleration g=9.8 "Gravitational acceleration";

  parameter Real c(unit="1")=0.8 "Elastic bouncing coeff.";
  parameter Real coef(unit="1")=1e-3 "Coeff. transition to StoppedOnFloor";

  SI.Distance x(start=10, fixed=true) "Vertical distance of ball to the floor";
  SI.Velocity v(start=0, fixed=true) "Ball velocity";
  Real eInicial(unit="J/kg") "Initial specific energy";
  Boolean parada(start=false, fixed=true)  "Ball in StoppedOnFloor mode?";

equation

  der(v) = if parada then 0 else -g;
  der(x) = v;

  when initial() then
    eInicial = 0.5*v^2 + g*x;
  end when;

  when x < 0 then
    assert(0.5*(c*v)^2 > g*2*eveps, "Model out of its experimental frame");
    parada = (0.5*(c*v)^2 < coef*eInicial);
    reinit(v, if parada then 0 else -c*v);
  end when;

  when parada then
    reinit(x, 0);
  end when;

end BouncingBall3;
```

Modelica Code 9.4: Bouncing ball model shown in Figure 9.11.

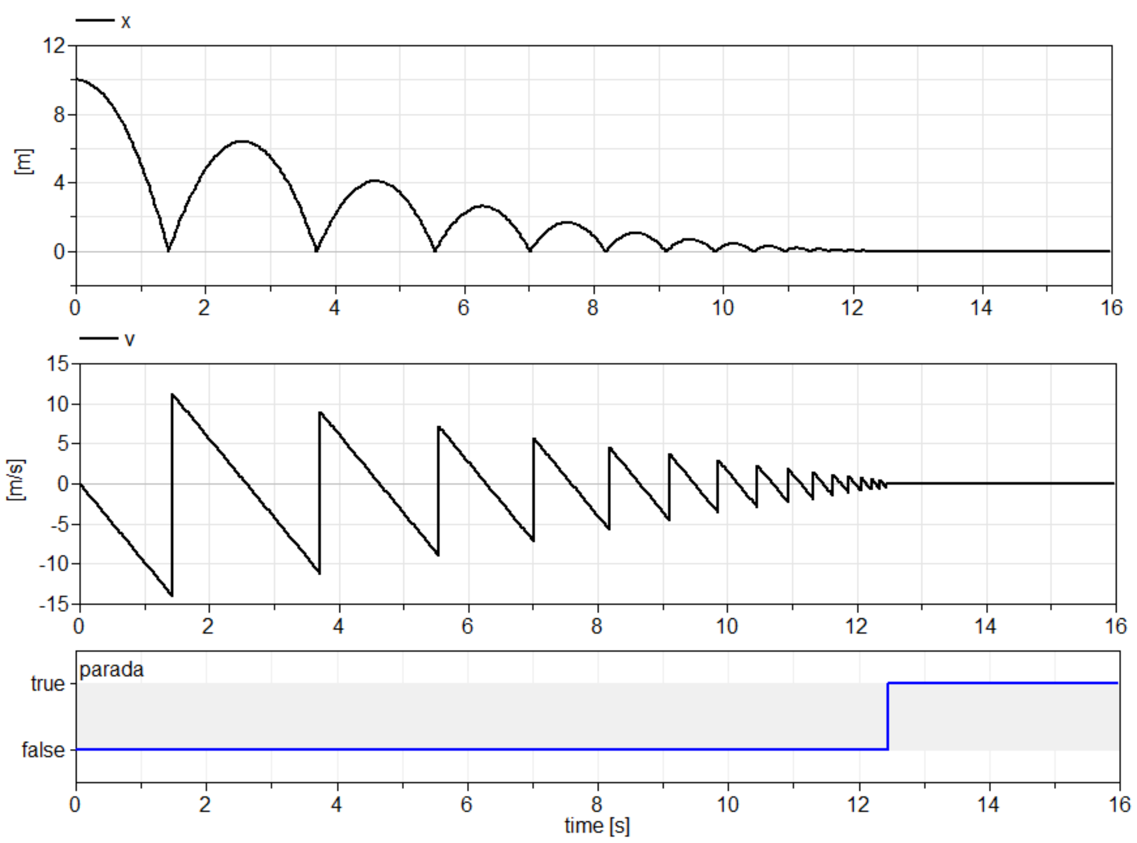


Figure 9.12: Result obtained simulating Modelica Code 9.4.

– The initial specific energy is computed using a when clause, whose condition is the initial() function. A value is assigned to the `eInicial` variable at the start time, and this value remains constant during the simulation.

– An assert sentence is used to check whether the model has gone out of its experimental frame.

– The model has two state variables in both modes. When the transition to the *StoppedOnFloor* mode takes place, the x and v state variables are reinitialized to zero, and the time derivatives of both variables are zero while the model remains in this mode. So, the ball position and velocity are zero while in the *StoppedOnFloor* phase.

9.6 Dry fiction

In this section, we are going to model the friction force between two contacting solid objects. The proposed model is described by the characteristic curve shown in

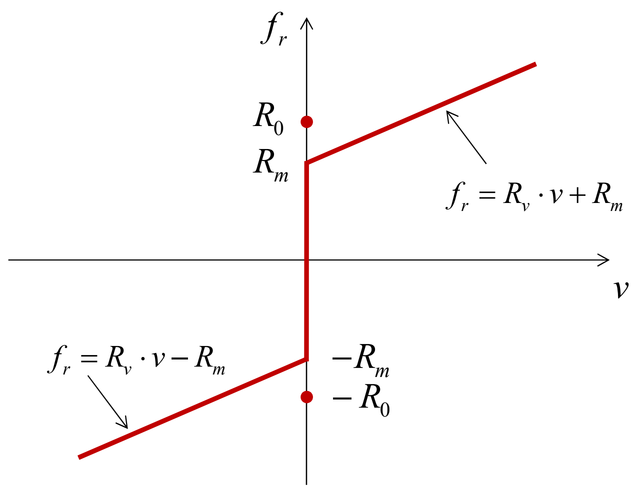


Figure 9.13: Characteristic curve of the dry friction force.

Figure 9.13, which depends on three parameters: R_0, R_m and R_v. This curve has the following meaning.

- While the relative velocity (v) between the two objects is different from zero, the friction force (f_r) depends linearly on this relative velocity.

$$f_r = \begin{cases} R_v \cdot v - R_m & \text{if} \quad v < 0 \\ R_v \cdot v + R_m & \text{if} \quad v > 0 \end{cases} \tag{9.26}$$

- Suppose that the relative velocity between the two objects is zero, and an external force is applied. The friction force is modeled as follows.

  - If the module of the external force is not larger than a certain value R_0, the friction force exactly counteracts the applied force, so that the objects remain in relative rest.
  - If the module of the external force is larger than R_0, then the friction force is not able to avoid the relative movement of the objects, and the friction force is described by Eq. (9.26).

This behavior of the friction force can be described using the following two-mode model.

1. In this mode, the velocity between the two bodies is different from zero, and the friction force is related to the relative velocity by Eq. (9.26). The transition condition to leave this mode is the relative velocity to become zero.

2. In this mode, that we name *Stuck*, the relative velocity between the objects is zero. The friction force has the value that makes the relative velocity to continue being zero. The friction force is this mode, named f_c, is calculated by imposing $v = 0$. The transition condition to leave this mode is $|f_c| > R_0$.

The model in the *Stuck* mode has one more equation ($v = 0$) and one more variable (f_c) than in the other mode. As Modelica does not allow to declare variables local to a particular mode, a value has to be assigned to the f_c variable while the model is not in the *Stuck* mode. This value is arbitrary, because f_c only has a physical meaning while the model is in the *Stuck* phase. By simplicity, we set this variable to zero. Therefore, the friction force is described by the following constitutive relationship.

$$\begin{aligned} f_r &= \textbf{if} \quad v > 0 \quad \textbf{then} \quad R_v \cdot v + R_m \\ & \textbf{elseif} \quad v < 0 \quad \textbf{then} \quad R_v \cdot v - R_m \\ & \qquad\qquad\qquad \textbf{else} \quad f_c \\ 0 &= \textbf{if} \quad \textit{Stuck} \quad \textbf{then} \quad v \\ & \qquad\qquad\qquad \textbf{else} \quad f_c \end{aligned} \tag{9.27}$$

However, the model should not be formulated in this way. The reason is that, in mechanical system models, velocities are typically either state variables or calculated from state variables. In consequence, the friction model in the *Stuck* mode, by imposing $v = 0$, reduces the number of DoF of the complete model. Let's explore another approach.

As accelerations typically are not state variables of mechanical system models, let's replace the equation $v = 0$ by $\frac{dv}{dt} = 0$. This is, instead of imposing the relative velocity to be zero while in the *Stuck* phase, it is imposed the relative acceleration to be zero. In addition, when the model enters in the *Stuck* phase, the relative velocity is reinitialized to zero. To goal is to avoid the numeric drifting of the relative displacement, which may become relevant if the model stays in the *Stuck* mode for a long time.

The next step is to specify the transition conditions among the model modes. We suppose that the model can be in any of the following five modes:

– *Stuck*. The relative velocity is zero while in this mode, and the friction force satisfies $-R_0 \leq f_r \leq R_0$.

– *Forward* and *Backward*. The relative velocity between the bodies is different from zero. In the *Forward* mode, the relative velocity satisfies $v > 0$ and

Table 9.1: Modes of the dry friction model.

Mode	**Condition**
Forward	$v > 0$ **and** $f_r = R_v \cdot v + R_m$
StartForward	$v = 0$ **and** $a > 0$ **and** $f_r = R_m$
Stuck	$v = 0$ **and** $a = 0$ **and** $-R_0 \leq f_r \leq R_0$
StartBackward	$v = 0$ **and** $a < 0$ **and** $f_r = -R_m$
Backward	$v < 0$ **and** $f_r = R_v \cdot v - R_m$

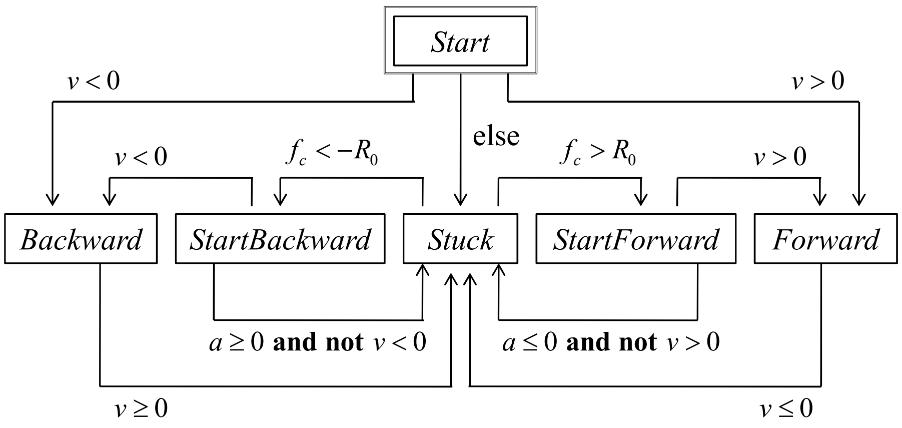


Figure 9.14: Transitions among the modes of the dry friction model.

the friction force is $f_r = R_v \cdot v + R_m$. In the *Backward* phase, $v < 0$ and $f_r = R_v \cdot v - R_m$.

– *StartForward* and *StartBackward*. These are intermediate modes between relative resting and sliding. While in these modes, the relative speed is zero, but the acceleration is different from zero. The friction force in the *StartForward* mode is $f_r = R_m$, and in the *StartBackward* mode is $f_r = -R_m$.

The conditions to stay in each mode are summarized in Table 9.1. The transitions among the modes can be described by means of a finite state machine, as in Figure 9.14. The diagram has six modes: the five previously described and the *Start* mode, in which the model is at the initialization.

Finite state machines can be described in Modelica by declaring a Boolean variable for each mode, and describing the condition for staying in each mode in the following way:

$$\begin{array}{rlll} mode = & \mathbf{pre}(modePre1)\ \mathbf{and}\ conditionIn1 & \mathbf{or} & \\ & \mathbf{pre}(modePre2)\ \mathbf{and}\ conditionIn2 & \mathbf{or} & \\ & \ldots & \mathbf{or} & \\ & \mathbf{pre}(mode)\ \mathbf{and\ not} & (\ conditionOut1\ \mathbf{or} & \\ & & \ \ conditionOut2\ \ldots\ ) & \end{array} \tag{9.28}$$

where $modePre1$, $modePre2$, ... are the modes from which a transition to $mode$ can occur if the $conditionIn1$, $conditionIn2$, ... conditions are fulfilled, respectively. The Boolean expressions $conditionOut1$, $conditionOut2$, ... are the conditions for the exit transitions from $mode$.

The Modelica description of the finite state machine shown in Figure 9.14 is the following.

```
Forward = pre(Start)        and v>0 or
          pre(StartForward) and v>0 or
          pre(Forward)      and not v<=0;
Backward = pre(Start)         and v<0 or
           pre(StartBackward) and v<0 or
           pre(Backward)      and not v>=0;
StartForward = pre(Stuck)        and fc > R0 or
               pre(StartForward) and not (v>0 or a<=0 and not v>0);
StartBackward = pre(Stuck)         and fc < -R0 or
                pre(StartBackward) and not (v<0 or a>=0 and not v<0);
Start = if initial() then true else false;
Stuck = not ( Forward      or Backward or StartForward or
              StartBackward or Start );
```

To illustrate the application of the friction model, a Modelica library named *Lib-Friction* has been programmed. See Modelica Code 9.5 – 9.7. The library architecture is shown in Figure 9.15.

The *Port* connector describes the mechanical port, that is composed of two variables: the position and the force. The velocity could have been chosen as across variable, instead of the position.

The *Transbody* model describes an object that can move in one dimension. The object has only one connector. The force variable of the connector is the net force applied to the object, which accelerates it. The position variable of the connector represents the position where the object is.

The *Inertial* model describes an object fixed in the origin of coordinates. This model will be used to represent the floor.

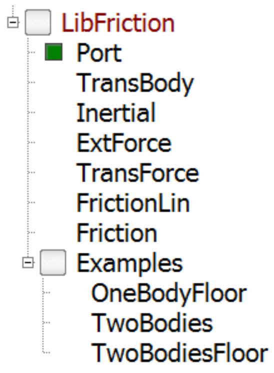

LibFriction
Port
TransBody
Inertial
ExtForce
TransForce
FrictionLin
Friction
Examples
OneBodyFloor
TwoBodies
TwoBodiesFloor

Figure 9.15: Architecture of the *LibFriction* library.

```
package LibFriction

import SI = Modelica.SIunits;

connector Port
   SI.Position p;
   flow SI.Force f;
end Port;

model TransBody
   Port port;
   parameter SI.Mass m = 1;
   parameter SI.Position p_Initial = 0;
   parameter SI.Velocity v_Initial = 0;
   SI.Position p( start=p_Initial, fixed=true);
   SI.Velocity v( start=v_Initial, fixed=true);
   SI.Acceleration a;
equation
   port.f = m*a;
   port.p = p;
   der(p) = v;
   der(v) = a;
end TransBody;

model Inertial
   Port port;
equation
   port.p = 0;
end Inertial;

model ExtForce
   Port port;
   SI.Force f;
   parameter Real Tstart = 50;
   parameter Real Tend = 75;
   parameter Real Kf = 1;
equation
   f = if time > Tstart and time < Tend
       then Kf*(time-Tstart)
       else 0;
   port.f = -f;
end ExtForce;
```

Modelica Code 9.5: *LibFriction* library (1/3).

```
model TransForce
   Port port1;
   Port port2;
   SI.Position p;
   SI.Velocity v(stateSelect = StateSelect.always);
   SI.Acceleration a;
equation
   p = port2.p - port1.p;
   der(p) = v;
   der(v) = a;
end TransForce;

model FrictionLin
   extends TransForce;
   SI.Force fr "Friction force";
   parameter Real Kf=1;
equation
   port2.f = -fr;
   port1.f = fr;
   fr = -Kf*v;
end FrictionLin;

model Friction
   extends TransForce;
   SI.Force fr "Friction force";
   SI.Force fc "Dummy variable";
   parameter Real R0 = 1 "Threshold value";
   parameter Real Rm = 0.5;
   parameter Real Rv = 1;
   Boolean Start(start=true, fixed=true);
   Boolean Stuck(start=false, fixed=true);
   Boolean StartBackward(start=false, fixed=true);
   Boolean StartForward(start=false, fixed=true);
   Boolean Backward(start=false, fixed=true);
   Boolean Forward(start=false, fixed=true);
   constant SI.Mass unitMass = 1;
equation
   port2.f = -fr;
   port1.f = fr;
   fr = if Forward then -(Rv*v+Rm) else
        if Backward then -(Rv*v-Rm) else
        if StartForward then -Rm else
        if StartBackward then Rm else fc;
   0 = if Stuck then unitMass*a else fc;

   Stuck = not (Forward or Backward or StartForward or
                StartBackward or Start);
   Forward = pre(Start) and v>0 or
             pre(StartForward) and v>0 or
             pre(Forward) and not v<=0;
   Backward = pre(Start) and v<0 or
              pre(StartBackward) and v<0 or
              pre(Backward) and not v>=0;
   StartForward = pre(Stuck) and fc < -R0 or
                  pre(StartForward) and not (v>0 or a<=0 and not v>0);
   StartBackward = pre(Stuck) and fc > R0 or
                   pre(StartBackward) and not (v<0 or a>=0 and not v<0);
   Start = if initial() then true else false;
   // reinit() has effect because v is state variable
   //See the value of the stateSelect attribute in the TransForce model
   when Stuck then
      reinit(v,0);
   end when;
end Friction;
```

Modelica Code 9.6: *LibFriction* library (2/3).

```
package Examples

model OneBodyFloor
   TransBody body(p_Initial=0, v_Initial=0, m=10);
   Friction friction;
   Inertial floor;
   ExtForce extForce1(Tstart=10, Tend=50, Kf=0.1);
   ExtForce extForce2(Tstart=100, Tend=150, Kf=-0.1);
equation
   connect(body.port, friction.port2);
   connect(friction.port1, floor.port);
   connect(extForce1.port, body.port);
   connect(extForce2.port, body.port);
   annotation (experiment(StopTime=250));
end OneBodyFloor;

model TwoBodies
   TransBody body1(p_Initial=0, v_Initial=2, m=10);
   TransBody body2(p_Initial=0, v_Initial=0, m=10);
   Friction friction;
   ExtForce extForce_c2(Tstart=100, Tend=150, Kf=-0.1);
equation
   connect( body1.port, friction.port2);
   connect( friction.port1, body2.port);
   connect( extForce_c2.port, body2.port);
   annotation (experiment(StopTime=250));
end TwoBodies;

model TwoBodiesFloor
// body2 - body1 - floor
   TransBody body1(p_Initial=0, v_Initial=2, m=10);
   TransBody body2(p_Initial=0, v_Initial=-2, m=10);
   Friction friction_c2_c1;
   Friction friction_c1_floor;
   Inertial floor;
   ExtForce extForce_c1(Tstart=50, Tend=100, Kf=0.1);
   ExtForce extForce_c2(Tstart=200, Tend=250, Kf=-0.1);
equation
   connect( body2.port, friction_c2_c1.port2);
   connect( friction_c2_c1.port1, body1.port);
   connect( body1.port, friction_c1_floor.port2);
   connect( friction_c1_floor.port1, floor.port);
   // External forces
   connect( extForce_c1.port, body1.port);
   connect( extForce_c2.port, body2.port);
   annotation (experiment(StopTime=350));
end TwoBodiesFloor;

end Examples;

end LibFriction;
```

Modelica Code 9.7: *LibFriction* library (3/3).

The *ExtForce* model describes an external force that can be applied to an object. The model has three parameters: T_{start}, T_{end} and K_f. The force is zero outside the $(T_{start}, T_{end}]$ time interval. Within this time interval, the force increases linearly with slope K_f N/s, stating from the zero value at the T_{start} time.

The *TransForce* model has two mechanical ports, and defines the relative position and velocity of the two ports. This model will be used as a superclass of the friction models.

The library contains two different models of the friction. In the *FrictionLin* model, the friction force depends linearly on the relative velocity for any value of the relative velocity. The *Friction* model describes the six-mode friction model shown in Figure 9.14, whose characteristic curve is shown in Figure 9.13.

The *Examples* package includes three models. The *OneBodyFloor* model describes the sliding motion with dry friction of a body over the floor. An external force is applied to the body during two different time intervals. The *TwoBodies* model describes the sliding motion with dry friction of two bodies. Additionally, the *TwoBodiesFloor* model includes the floor model, so that the sliding with friction occurs between the two bodies, and between the lower body and the floor. The simulation of these examples is left to the reader.

9.7 Heat conduction in a wall

The heat conduction in the wall of a cooling chamber is analyzed in this section. The wall is composed of three layers of different materials, named A, B and C, with width $L_A = 15$ mm, $L_B = 100$ mm and $L_C = 75$ mm respectively. A transversal cut of the wall is represented in the upper part of Figure 9.16.

The inner surface of the wall is at a constant temperature of 0 ^{0}C, while the outer temperature varies between -20 ^{0}C and 20 ^{0}C along 24 hours. The temperatures of the inner and outer surfaces of the wall, T_1 and T_5, are described by Eqs. (9.29) and (9.30), where time (t) is expressed in seconds (24 hours are equivalent to $24 \cdot 60 \cdot 60 = 86400$ seconds), and temperature in Kelvin.

$$T_1 = 273.15 \tag{9.29}$$

$$T_5 = 273.15 + 20 \cdot \sin\left(\frac{2 \cdot \pi \cdot t}{86400}\right) \tag{9.30}$$

The goal is to calculate the heat flow rate per unit of cross-sectional area, and the temperatures at the interfaces between the different materials (T_2, T_3) and at the wall's outer surface (T_4). To this end, the system is described as the equivalent thermal circuit shown in the lower part of Figure 9.16. The S parameter that appears in the thermal resistance denominator is the wall surface. This parameter is set to $S = 1$ m^2, so that Q is equal to the heat flow rate per unit of cross-sectional area. The thermal conductivities, κ_A, κ_B and κ_C, are expressed in W·m^{-1}·K^{-1} in Eqs. (9.31) – (9.33).

$$\begin{aligned} \kappa_A &= 0.151 && (9.31)\\ \kappa_B &= 2.5 \cdot \exp\left(\frac{-1225}{T}\right) && (9.32)\\ \kappa_C &= 0.762 && (9.33) \end{aligned}$$

The thermal conductivity of the B material is a function of the temperature (T, expressed in Kelvin), as can be seen in Eq. (9.32). In first approximation, the temperature of the B material is assumed to be the average of the temperatures in the A-B and B-C interfaces. This is,

$$\kappa_B = 2.5 \cdot \exp\left(\frac{-1225}{0.5 \cdot (T_2 + T_3)}\right) \tag{9.34}$$

The natural convection heat transfer coefficient, expressed in W·m^{-2}·K^{-1}, is calculated as follows.

$$\mathrm{h} = 1.37 \cdot \left|\frac{T_5 - T_4}{6}\right|^{1/4} \tag{9.35}$$

Naming Q to the heat flow rate (watts) from the inner to the outer surface of the wall, the constitutive relationships of the four thermal resistors are:

$$\begin{aligned} T_1 - T_2 &= Q \cdot \frac{L_A}{\kappa_A \cdot S} && (9.36)\\ T_2 - T_3 &= Q \cdot \frac{L_B}{\kappa_B \cdot S} && (9.37)\\ T_3 - T_4 &= Q \cdot \frac{L_C}{\kappa_C \cdot S} && (9.38)\\ T_4 - T_5 &= Q \cdot \frac{1}{\mathrm{h} \cdot \mathrm{S}} && (9.39) \end{aligned}$$

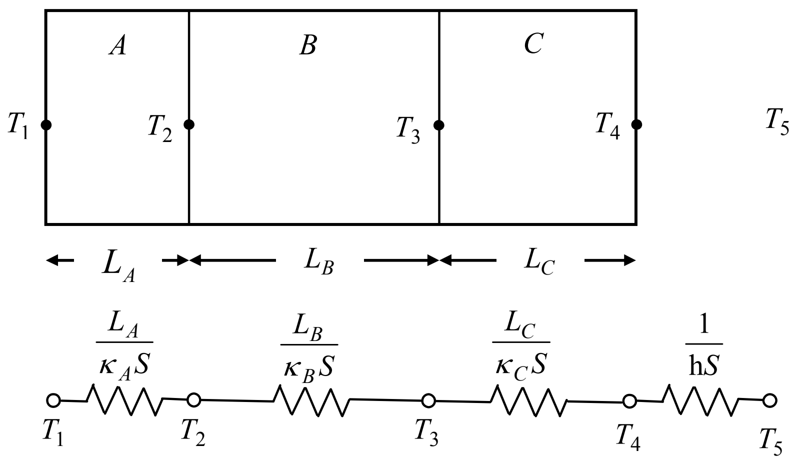


Figure 9.16: Structure of the chamber wall (above) and equivalent thermal circuit (below).

```
model MultilayerWall
  import SI = Modelica.SIunits;
  // Thickness of the layers
  parameter SI.Length La = 15E-3;
  parameter SI.Length Lb = 100E-3;
  parameter SI.Length Lc = 75E-3;
   // Cross-sectional area of the wall
  parameter SI.Area S =  1;
  // Thermal conductivities of the layers
  parameter SI.ThermalConductivity Ka = 0.151;
  SI.ThermalConductivity Kb;
  parameter SI.ThermalConductivity Kc = 0.762;
  // Convective heat transfer coefficient
  SI.CoefficientOfHeatTransfer h;
  // Temperatures
  parameter SI.Temperature T1 = 273.15;
  SI.Temperature T2(start=273,fixed=false);
  SI.Temperature T3(start=273,fixed=false);
  SI.Temperature T4(start=273,fixed=false);
  SI.Temperature T5;
  // Heat flow rate
  SI.HeatFlowRate Q;
equation
  // Outer temperature
  T5=273.15+20*sin(time*2*Modelica.Constants.pi/86400);
  // Thermal conductivity of the B material
  Kb= 2.5*exp(-1225*2/(T2+T3));
  // Convective heat transfer coefficient
  h= if noEvent(T4>T5) then 1.37*((T4-T5)/6)^0.25 else
          if noEvent(T5>T4) then 1.37*((T5-T4)/6)^0.25 else
          1;
  // Constitutive relationships of the thermal resistors
  T1 - T2 = Q * La/(Ka*S);
  T2 - T3 = Q * Lb/(Kb*S);
  T3 - T4 = Q * Lc/(Kc*S);
  T4 - T5 = Q * 1 /(h*S);
end MultilayerWall;
```

Modelica Code 9.8: Heat transfer in the three-layer wall shown in Figure 9.16.

In summary, the model has 8 variables (T_1, T_2, T_3, T_4, T_5, Q, κ_B, h) and is composed of 8 equations. These are Eqs. (9.29), (9.30), (9.34), (9.35) and (9.36) – (9.39). The L_A, L_B, L_C, S, κ_A and κ_C parameters have known values.

Observe Eq. (9.39). The convection heat transfer coefficient, h, appears in the denominator of the thermal resistance. When the T_4 and T_5 temperatures become equal, the value of h, calculated from the Eq. (9.35), is zero. The value h = 0 implies an infinite value of the thermal resistance, 1/(h · S), which generates a runtime numerical error.

To avoid this error, let's assign to h a value different from zero when T_4 and T_5 are equal. As the heat flow rate in this situation is $Q = 0$, it does not matter the value assigned to h when $T_4 = T_5$, as long as it is different from zero. We define h as follows:

$$\begin{array}{rl} \mathrm{h} = & \textbf{if noEvent}(T4 > T5) \textbf{ then } 1.37 \cdot ((T4 - T5)/6)\hat{}0.25 \\ & \textbf{else if noEvent}(T5 > T4) \textbf{ then } 1.37 \cdot ((T5 - T4)/6)\hat{}0.25 \\ & \textbf{else } 1; \end{array} \tag{9.40}$$

Executing the simulation of Modelica Code 9.8 during 86400 s, the result shown in Figure 9.17 is obtained. The biggest thermal step occurs in the B material, as shown in the upper plot of the figure. The heat flow rate Q is positive while it goes out from the cooling chamber, i.e., while $T_1 > T_5$.

Remember that the thermal conductivity of the B material is a function of the temperature, and that we have assumed the temperature of the B material to be equal to the average of the temperatures at the A-B and B-C interfaces. To estimate the error associated to this modeling hypothesis, the B material is split into N_{elemB} equal layers, each one with a thickness equal to L_B/N_{elemB}. We define a vector of temperatures, whose components are:

$$T_B[1], \ldots, T_B[N_{elemB} + 1] \tag{9.41}$$

so that

$$T_B[1] = T_2 \tag{9.42}$$

$$T_B[N_{elemB} + 1] = T_3 \tag{9.43}$$

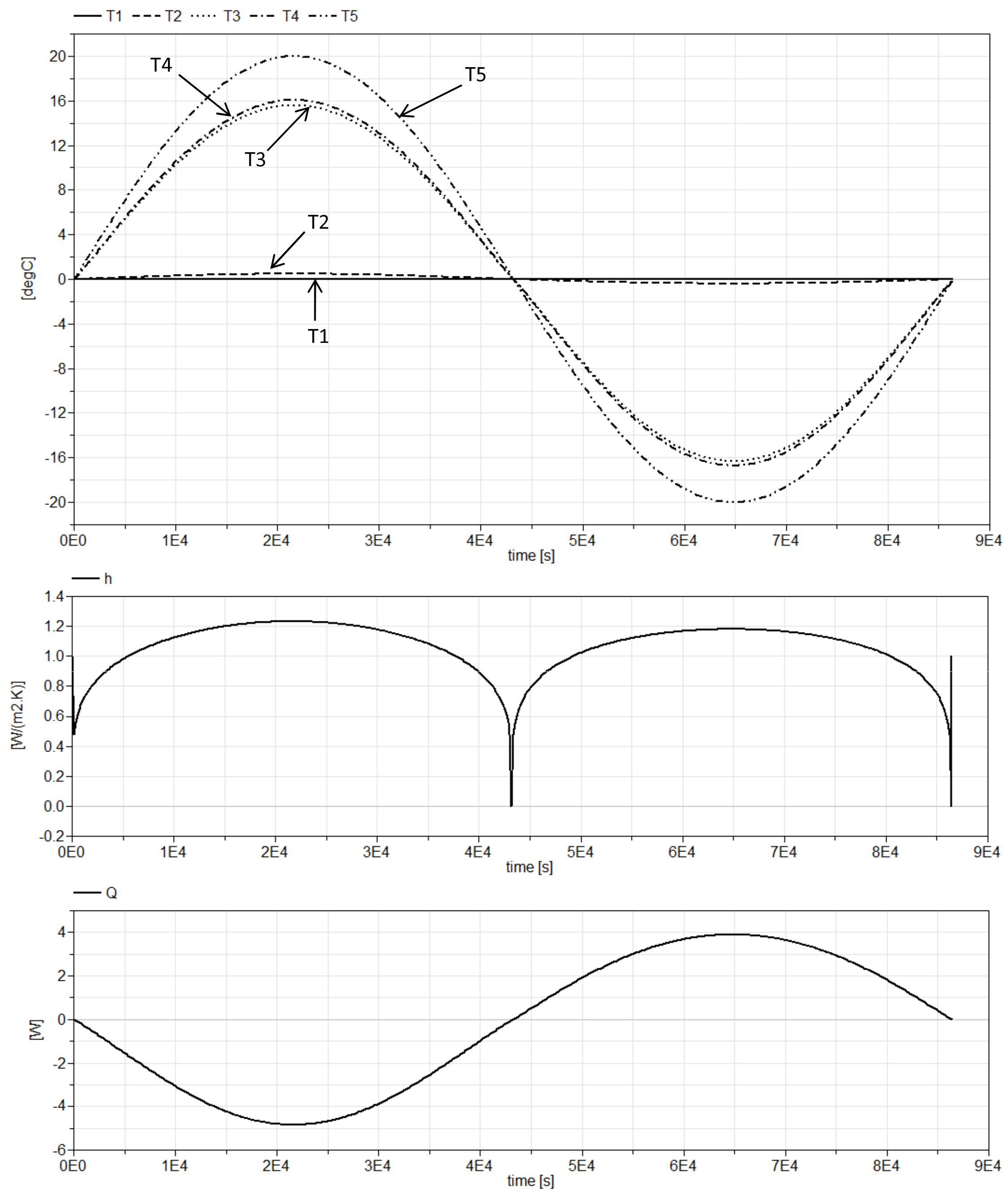


Figure 9.17: Result obtained simulating Modelica Code 9.8.

and where

$$T_B[i] \qquad \text{with} \;\; i = 2, \ldots, N_{elemB} \tag{9.44}$$

is the temperature at the interface between the $(i-1)$-th and the i-th layer in which the B material has been split. The thermal conductivity at the i-th layer of the B material is:

$$\kappa_B[i] = 2.5 \cdot \exp\left(\frac{-1225}{0.5 \cdot (T_B[i] + T_B[i+1])}\right) \tag{9.45}$$

The constitutive relationship of the thermal resistor describing the i-th layer of the B material is:

$$T_B[i] - T_B[i+1] = Q \cdot \frac{L_B/N_{elemB}}{\kappa_B[i] \cdot S} \tag{9.46}$$

The model is described in Modelica Code 9.9. The simulation has been performed by dividing the B material into $N_{elemB} = 50$ layers, and the obtained results have been compared with the results obtained by simulating Modelica Code 9.8. The result of this comparison is shown in Figure 9.18, where can be seen the order of magnitude of the error in the heat flow rate, and the T_2 and T_3 temperatures.

```
model NMultilayerWall
  import SI = Modelica.SIunits;
  // Number of layers of the B material
  constant Integer NelemB = 50;
  // Thickness of the layers
  parameter SI.Length La = 15E-3;
  parameter SI.Length Lb = 100E-3;
  parameter SI.Length Lc = 75E-3;
   // Wall cross-sectional area
  parameter SI.Area S =  1;
  // Thermal conductivities of the layers
  parameter SI.ThermalConductivity Ka = 0.151;
  SI.ThermalConductivity Kb[NelemB];
  parameter SI.ThermalConductivity Kc = 0.762;
  // Coefficient of convection heat transfer
  SI.CoefficientOfHeatTransfer h;
  // Temperatures
  parameter SI.Temperature T1 = 273.15;
  SI.Temperature T2(start=273,fixed=false);
  SI.Temperature T3(start=273,fixed=false);
  SI.Temperature T4(start=273,fixed=false);
  SI.Temperature T5;
  SI.Temperature TB[NelemB+1]( start=273*ones(NelemB+1),fixed=false);
  // Heat flow rate
  SI.HeatFlowRate Q;
equation
  // Outer temperature
  T5=273.15+20*sin(time*2*Modelica.Constants.pi/86400);
  // Coeff. of convection heat transfer
  h= if noEvent(T4>T5) then 1.37*((T4-T5)/6)^0.25 else
          if noEvent(T5>T4) then 1.37*((T5-T4)/6)^0.25 else
          1;
  // Constitutive relationships of the thermal resistors
  T1 - T2 = Q * La/(Ka*S);
  T3 - T4 = Q * Lc/(Kc*S);
  T4 - T5 = Q * 1 /(h*S);
  // Layer B
  TB[1] = T2;
  for i in 1:NelemB loop
     TB[i] - TB[i+1] = Q * (Lb/NelemB)/(Kb[i]*S);
     Kb[i] = 2.5*exp(-1225*2/(TB[i] + TB[i+1]));
  end for;
  TB[NelemB+1] = T3;
end NMultilayerWall;
```

Modelica Code 9.9: Heat transfer in the wall, with the B material divided into layers.

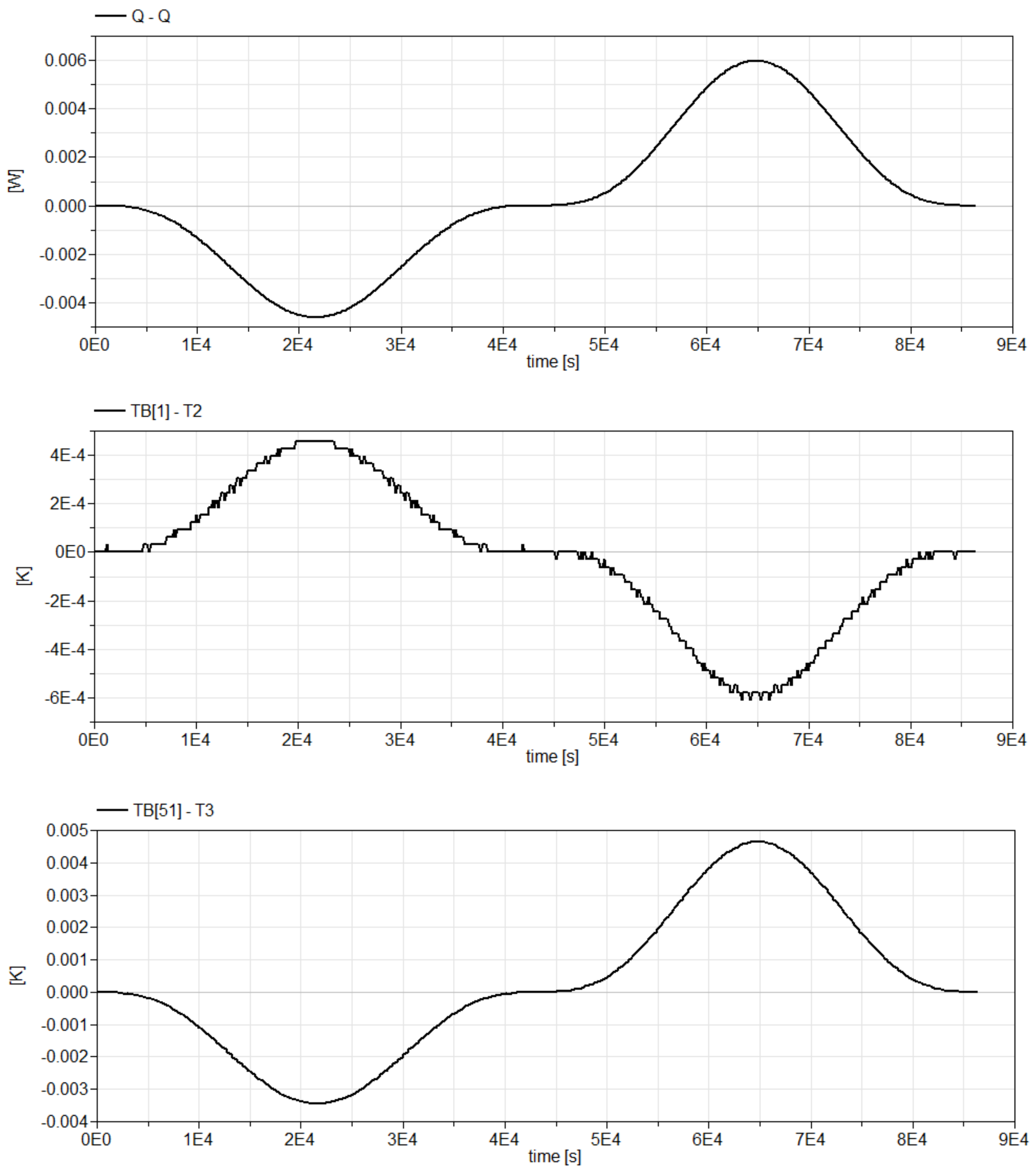


Figure 9.18: Comparison of the results obtained with $N_{elemB} = 50$ and $N_{elemB} = 1$.

9.8 Further reading

Some of the examples discussed in this lesson have been extracted from bibliography. Modeling and simulation of an ideal diode is discussed in (Elmqvist et al. 2001). The dry friction model, implemented using the Dymola language, is described in (Elmqvist et al. 1993). Although the Dymola language is no longer in use, the event detection procedure described in this article is essentially the same as the procedure employed by the Modelica modeling environments nowadays. The model of heat conduction in a wall is described in (Cutlip & Shacham 1999).

Subject Index

algebraic loop, 171
  manipulation, 173
algorithm, 16, 112

chattering, 238
computational causality, 15, 112

DAE
  DASSL, 181
  high index, 136, 145
  index, 136
  initialization, 147
  singular, 115
DASSL, 181
degrees of freedom, 116, 128
diode, 40
  ideal, 253
DoF, 128
dot notation, 75
Dymola
  button
    Check, 27
    Modelica Text, 25
    Simulate, 31
    Store in Model, 30
    Translate, 27
  Change directory, 25
  communication
    interval, 240
  event, 227
    eveps, 231
  file
    dsin.txt, 31
    dsmodel.c, 27
    dymosim.exe, 27
    result, 240
  New model, 25
  window
    Simulation setup, 28
  working directory, 25

equation, 15, 112
  instantaneous, 247
  trivial, 20
event, 22, 187, 243, 244
  chain, 224, 225
  condition, 188
  crossing function, 230
  detection, 230, 246
  Dymola, 227
  endogenous, 236
  execution, 188, 225
  execution order, 225
  exogenous, 236
  iteration, 236
  noEvent, 237
  reinit, 189, 224
  state, 236
  state event, 22

time, 236
time event, 22
trigger, 224
trigger condition, 22

function
sign, 237

information encapsulation, 17
initialization, 211
integration
Adams-Bashforth, 178
Adams-Moulton, 178
BDF, 180, 181
DASSL, 181
Euler's method, 125
explicit method, 177
inline, 182
method order, 177
mixed mode, 183
RKF45, 180
Runge-Kutta, 178
step length, 176
interface, 71
interpolation, 64
invariant, 187

law
gravitation, 55
of Newton, 49, 84
Ohm, 15

matrix
BLT, 116
incidence, 113
singular, 137
structural Jacobian, 113
model
flat, 20
hybrid, 243
mode, 201
overdetermined, 116
partition, 116
underdetermined, 116, 118
variable structure, 23, 201, 236
Modelica
.mo, 80
.mos, 30
abs, 64
algorithm, 64
annotation, 18, 105
diagram, 18
icon, 18
info, 18
array
declaration, 55
functions, 57
index, 57
assert, 44, 66, 262
attribute
displayUnit, 44, 64
final, 45
fixed, 24, 44
max, 44
min, 44
quantity, 44
start, 24, 44
stateSelect, 136
unit, 44, 64
block, 18
Boolean, 44
built-in function, 64
class
partial, 17, 33
code comments, 24
component vector, 88
connect, 80
connector, 17, 18, 71
unconnected, 80
constant, 44

cross, 58
der(), 24
diagonal, 58
discrete, 44
div, 64
event, 16, 244
extends, 72
external function, 16
fill, 58
fixed, 169, 212, 214
flow, 71
for, 57
for with iterators, 57
function, 16, 18, 64
  internal memory, 64
identity, 58
if, 74, 202, 222, 244
import, 79
information encapsulation, 17
inheritance, 17, 72
  import, 80
initial, 216, 219, 222, 249
initial algorithm, 216
initial equation, 89, 216
initialization, 211
inner, 106
Integer, 44
linspace, 58
LogVariable, 251
max, 58
message writing, 251
min, 58
model, 18
  save, 80
MODELICAPATH, 81
modularity, 17
MSL
  Constants, 45
  SIunits, 45
ndims, 58
noEvent, 237, 246
ones, 58
outer, 106
package, 18, 75, 80
  encapsulated, 79
package.mo, 81
parameter, 17, 44
  by-default value, 73
parametrization, 17
partial model, 72
physical field, 106
pre(), 244
product, 58
protected, 72, 85
public, 72
Real, 44
record, 18, 85
redeclare, 82
reinit, 167, 193, 248
rem, 64
replaceable, 81
sample, 249
script file, 30
sign, 64
simulation termination, 250
size, 58
sqrt, 64
start, 212, 214
state variable, 136
stateSelect, 169
String, 44, 250
  concatenation, 250
  integerString, 250
  realString, 250
sum, 58
terminal, 249
terminate, 250
time, 43

  transpose, 58
  type, 18, 45
  variable
    attribute, 44
    built-in types, 44
  vector, 55
  when, 247
  within, 81
  zeros, 58
modeling
  causal, 16
  environment, 20
  equation-based, 15
  non-causal, 15
  object-oriented, 17
  physical paradigm, 14

node, 19
numerical differentiation, 146

Omola, 186
OpenModelica, 32
  Check Model, 33
  OMEdit, 32
    Diagram View, 33
    Documentation View, 33
    Icon View, 33
    Options, 36
    Simulate, 36
    simulation interval, 36
    Simulation Setup, 36
    Text View, 33
    working directory, 36

Pantelides algorithm, 134, 154
partial class, 17
partition, 116
pendulum, 213
PI controller, 93, 216
pin, 19

simulation
  algorithm, 125
  hybrid model, 21
single-assignment rule, 61, 248
SPICE, 85, 174
stiff system, 171
structural singularity, 115, 122
switch
  ideal, 201
  resistive, 208

Tarjan's algorithm, 117
tearing, 175
temporal discretization, 176
time, 113

variable
  across, 17
  algebraic, 125
  alias, 20
  auxiliar, 113
  classification, 122
  effort, 17
  flow, 17
  parameter, 125
  state variable, 125, 157
  through, 17

References

Andersson, M. (1990), *Omola. An Object-Oriented Language for Model Representation*, Licenciate thesis TFRT-3208. Department of Automatic Control. Lund Institute of Technology. Lund. Sweden.

Andersson, M. (1994), *Object-oriented Modeling and Simulation of Hybrid Systems*, PhD Diss., Lund Institute of Tech., Sweden.

Åström, K. J., Elmqvist, H. & Mattsson, S. E. (1998), Evolution of continuous-time modeling and simulation, *in* '12th European Simulation Multiconference', Manchester, UK, pp. 9–18.

Åström, K. J. & Hagglund, T. (1995), *PID Controllers: Theory, Design and Tuning*, ISA Press.

Barton, P. (1992), *The Modelling and Simulation of Combined Discrete/Continuous Processes*, Ph.D. Thesis. Department of Chemical Engineering. Imperial College of Science, Technology and Medicine. London.

Bird, R. B., Stewart, W. E. & Lightfoot, E. N. (1975), *Transport Phenomena*, John Wiley & Sons.

Brenan, K. E., Campbell, S. L. & Petzold, L. R. (1996), *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM.

Cellier, F. E. (1979), *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, PhD Diss., Switzerland.

Cellier, F. E. (1991), *Continuous System Modeling*, Springer-Verlag.

Cellier, F. E., Elmqvist, H., Otter, M. & Taylor, J. H. (1993), Guidelines for modeling and simulation of hybrid systems, *in* 'IFAC World Congress', Sydney, Australia.

Cellier, F. E. & Kofman, E. (2006), *Continuous System Simulation*, Springer-Verlag.

Cellier, F., Otter, M. & Elmqvist, H. (1995), Bond graph modeling of variable structure systems, *in* '2nd Intl. Conference on Bond Graph Modeling and Simulation, ICBGM'95, Las Vegas', Las Vegas, USA, pp. 49–55.

Cutlip, M. B. & Shacham, M. (1999), *Problem Solving in Chemical Engineering with Numerical Methods*, Prentice-Hall.

Dassault Systèmes AB (2016), *Dymola. User Manual. Version Dymola 2017*, Dassault Systèmes AB, Lund, Sweden.

Dynasim AB (2004), *Dymola. User's Manual. Version 5.3a*, Dynasim AB, Lund, Sweden.

Elmqvist, H. (1978), *A Structured Model Language for Large Continuous Systems*, PhD Diss., Lund Institute of Tech., Sweden.

Elmqvist, H. (1993), Object-oriented modeling and automatic formula manipulation in Dymola, *in* 'SIMS'93, Scandinavian Simulation Society', Kongsberg, Norway.

Elmqvist, H., Cellier, F. E. & Otter, M. (1993), Object-oriented modeling of hybrid systems, *in* 'ESS'93, European Simulation Symposium', Delft, The Netherlands.

Elmqvist, H., Cellier, F. E. & Otter, M. (1994), Object-oriented modeling of power-electronic circuits using Dymola, *in* 'Proc. of CISS - First Joint Conference of Intl. Simulation Societies', Zurich, Switzerland.

Elmqvist, H., Cellier, F. E. & Otter, M. (1995), Inline integration: a new mixed symbolic/numeric approach for solving differential-algebraic equation systems, *in* 'ESM'95, SCS European Simulation MultiConference', Prague, Czech Republic.

Elmqvist, H., Mattsson, S. & Olsson, H. (2002), New methods for hardware-in-the-loop simulation of stiff models, *in* '2nd Intl. Modelica Conference', Oberpfaffenhofen, Germany, pp. 59–64.

Elmqvist, H., Mattsson, S. & Otter, M. (2001), 'Object-oriented and hybrid modeling in Modelica', *Journal Europeen des Systemes Automatises, APII – JESA* **35(1)**, 1–22.

Elmqvist, H. & Otter, M. (1994), Methods for tearing systems of equations in object-oriented modeling, *in* 'ESM'94, European Simulation Multiconference', Barcelona, Spain.

Franke, R., Casella, F., Otter, M., Sielemann, M., Elmqvist, H., Mattson, S. & Olsson, H. (2009), Stream connectors - an extension of Modelica for device-oriented modeling of convective transport phenomena, *in* '7^{th} Intl. Modelica Conference', Como, Italy.

Fritzson, P. (2011), *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*, Wiley.

Fritzson, P. (2015), *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3*, Wiley-IEEE Press.

Froment, G. & Bischoff, K. (1979), *Chemical Reactor Analysis and Design*, John Wiley & Sons.

Hogan, N. & Breedveld, P. (1995), *Integrated Modeling of Physical System Dynamics*, University of Twente. Report nr. 027R 95.

Incropera, F. & DeWitt, D. (1996), *Fundamentals of Heat and Mass Transfer*, Fourth Edition. John Wiley & Sons.

Karnopp, D. C., Margolis, D. L. & Rosenberg, R. C. (1990), *System Dynamics: A Unified Approach*, John Wiley & Sons.

Luyben, W. (1990), *Process Modeling, Simulation and Control for Chemical Engineers*, McGraw-Hill.

Mattsson, S. E., Elmqvist, H., Otter, M. & Olsson, H. (2002), Initialization of hybrid differential-algebraic equations in Modelica 2, *in* '2^{nd} Intl. Modelica Conference', Oberpfaffenhofen, Germany.

Mattsson, S. E., Olsson, H. & Elmqvist, H. (2000), Dynamic selection of states in Dymola, *in* 'Modelica Workshop 2000', Lund, Sweden.

Mattsson, S. E. & Söderlind, G. (1992), A new technique for solving high-index differential equations using dummy derivatives, *in* 'IEEE Symposium on Computer-Aided Control System Design', California, USA.

Mattsson, S. & Söderlind, G. (1993), 'Index reduction in differential-algebraic equations using dummy derivatives', *SIAM Journal on Scientific Computing* .

ModelicaTM (2000), *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial. Version 1.4*, Modelica Association.

ModelicaWebSite (2017), ‘Website of the Modelica Association’. http://www.modelica.org.

Olsson, H., Otter, M., Mattsson, S. & Elmqvist, H. (2008), Balanced models in Modelica 3.0 for increased model quality, *in* ‘6^{th} Intl. Modelica Conference’, Bielefeld, Germany, pp. 21–33.

OpenModelica (2017), ‘OpenModelica website’. www.openmodelica.org.

Otter, M. (2009), *Modeling, simulation and control with Modelica 3.1 and Dymola 7*.

Otter, M. & Olsson, H. (2002), New features in Modelica 2.0, *in* ‘2^{nd} Intl. Modelica Conference’, Oberpfaffenhofen, Germany, pp. 7.1–7.12.

Pantelides, C. (1988), ‘The consistent initialization of differential-algebraic systems’, *SIAM J. SCI. STAT. COMPUT.* **9**(2).

Ramirez, W. F. (1989), *Computational Methods for Process Simulation*, Butterworths.

Schiela, A. & Olsson, H. (2000), Mixed-mode integration for real-time simulation, *in* ‘Modelica Workshoop 2000’, Lund, Sweden.

Steward, D. V. (1981), *System Analysis and Management: Structure, Strategy and Design*, Petrocelli Books, Inc.

Thoma, J. U. (1990), *Simulation by Bondgraphs*, Springer-Verlag.

Tiller, M. (2001), *Introduction to Physical Modeling with Modelica*, Kluwer Academic Publishers Group.

Urquia, A. (2000), *Modelado Orientado a Objetos y Simulación de Sistemas Híbridos en el Ámbito del Control de Procesos Químicos*, PhD Diss., UNED, Madrid, Spain.

This book offers an introduction to the development and simulation of Modelica models for engineering applications. The target audience are bachelor's or master's level students, interested in modeling and simulation, and with a background in both physics and numerical methods. The book is structured into three parts. The modeling methodology and the Modelica features for continuous-time modeling are discussed in the first part of the book. The simulation of continuous-time Modelica models is addressed in the second part of the book. The third part of the book is devoted to discuss hybrid modeling and simulation in Modelica. The modeling methodology, the Modelica language features, and the use of modeling environments are explained through examples. This facilitates the use of this book in the context of student-centered learning strategies, such as problem-based learning, and project-based learning.

Alfonso Urquía and Carla **Martín** are professors in the Departamento de Informática y Automática, at the Universidad Nacional de Educación a Distancia (UNED) in Madrid, Spain; and members of the research group on Modelling & Simulation in Control Engineering of UNED. Further information is available at: www.euclides.dia.uned.es

This work has been developed within the project "InMotion - Innovative teaching and learning strategies in open modelling and simulation environment for student-centered engineering education", Project No. 573751-EPP-1-2016-1-DE-EPPKA2-CBHE-JP, co-funded by the Erasmus+ Programme of the European Union.

UNED Editorial

Juan del Rosal, 14
28040 MADRID
Tel. Dirección Editorial: 913 987 521

Co-funded by the Erasmus+ Programme of the European Union

Innovative teaching and learning strategies in open modelling and simulation environment for student-centered engineering education
573751-EPP-1-2016-1-DE-EPPKA2-CBHE-JP

InMotion