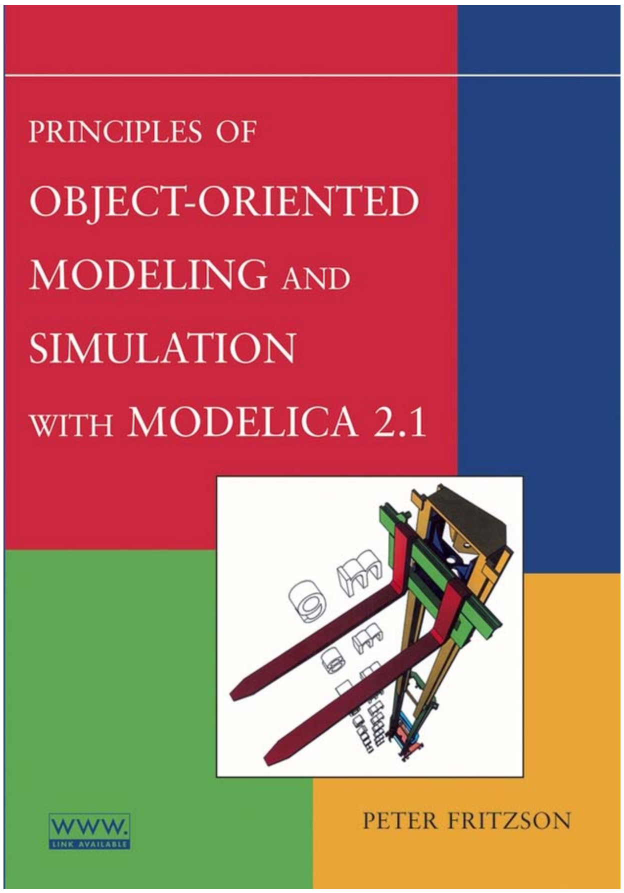
Peter Fritzson

Principles of Object-Oriented Modeling and Simulation with Modelica 2.1



Book reference:
Peter Fritzson: Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 939 pages,Wiley-IEEE Press, ISBN 0-471-471631.

Order, e.g. at www.Amazon.com,
in theUS: http://www.wiley.com/WileyCDA/WileyTitle/productCd-0471471631.html, or
outside: http://www.wileyeurope.com/WileyCDA/WileyTitle/productCd-0471471631.html

For further information visit: the book web page http://www.DrModelica.org, the Modelia Association web page http://www.modelica.org, the authors research page http://www.ida.liu.se/labs/pelab/modelica, or email the author at petfr@ida.liu.se

Preface

The Modelica modeling language and technology is being warmly received by the world community in modeling and simulation with major applications in virtual prototyping. It is bringing about a revolution in this area, based on its ease of use, visual design of models with combination of lego-like predefined model building blocks, its ability to define model libraries with reusable components, its support for modeling and simulation of complex applications involving parts from several application domains, and many more useful facilities. To draw an analogy—Modelica is currently in a similar phase as Java early on, before the language became well known, but for virtual prototyping instead of Internet programming.

About this Book

This book teaches modeling and simulation and gives an introduction to the Modelica language to people who are familiar with basic programming concepts. It gives a basic introduction to the concepts of modeling and simulation, as well as the basics of object-oriented component-based modeling for the novice, and a comprehensive overview of modeling and simulation in a number of application areas. In fact, the book has several goals:

- Being a useful textbook in introductory courses on modeling and simulation.
- Being easily accessable for people who do not previously have a background in modeling, simulation and objectorientation.
- Introducing the concepts of physical modeling, object-oriented modeling, and component-based modeling.
- Providing a complete but not too formal reference for the Modelica language.
- Demonstrating modeling examples from a wide range of application areas.
- Being a reference guide for the most commonly used Modelica libraries.

The book contains many examples of models in different application domains, as well as examples combining several domains. However, it is not primarily intended for the advanced modeler who, for example, needs additional insight into modeling within very specific application domains, or the person who constructs very complex models where special tricks may be needed.

All examples and exercises in this book are available in an electronic self-teaching material called DrModelica, based on this book, that gradually guides the reader from simple introductory examples and exercises to more advanced ones. Part of this teaching material can be freely downloaded from the book web site, `www.DrModelica.org`, where additional (teaching) material related to this book can be found, such as the exact version of the Modelica standard library (September 2003) used for the examples in this book. The main web site for Modelica and Modelica libraries, including the most recent versions, is the Modedica Association website, `www.Modelica.org`.

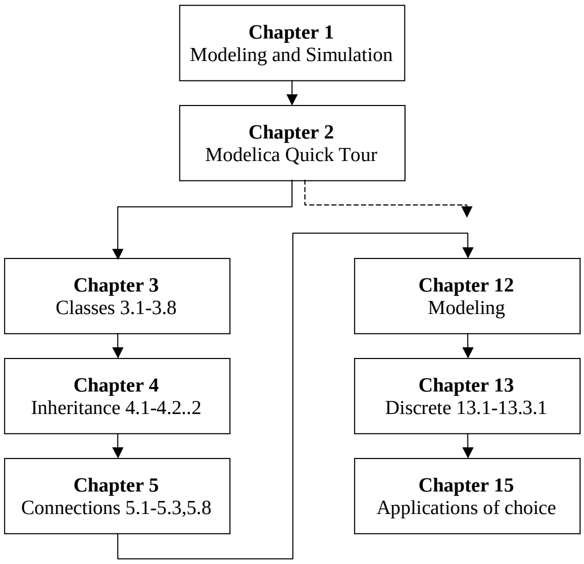
Reading Guide

This book is a combination of a textbook for teaching modeling and simulation, a textbook and reference guide for learning how to model and program using Modelica, and an application guide on how to do physical modeling in a number of application areas. The book can be read sequentially from the beginning to the end, but this will probably not be the typical reading pattern. Here are some suggestions:

- Very quick introduction to modeling and simulation – an object-oriented approach: Chapters 1 and 2.
- Basic introduction to the Modelica language: Chapter 2 and first part of Chapter 13.
- Full Modelica language course: Chapters 1–13.
- Application-oriented course: Chapter 1, and 2, most of Chapter 5, Chapters 12–15. Use Chapters 3–11 as a language reference, and Chapter 16 and appendices as a library reference.
- Teaching object orientation in modeling: Chapters 2–4, first part of Chapter 12.
- Introduction to mathematical equation representations, as well as numeric and symbolic techniques, Chapter 17-18.
- Modelica environments, with three example tools, Chapter 19.

An interactive computer-based self-teaching course material called DrModelica is available as electronic live notebooks. This material includes all the examples and exercises with solutions from the book, and is designed to be used in parallel when reading the book, with page references, etc.

The diagram below is yet another reading guideline, giving a combination of important language concepts together with modeling methodology and application examples of your choice. The selection is of necessity somewhat arbitrary – you should also take a look at the table of contents of other chapters and part of chapters so that you do not miss something important according to your own interest.



Acknowledgements

The members of the Modelica Association created the Modelica language, and contributed have many examples of Modelica code in the *Modelica Language Rationale* and *Modelica Language Specification* (see http://www.modelica.org), some of which are used in this book. The members who contributed to various versions of Modelica are mentioned further below.

First, thanks to my wife, Anita, who has supported and endured me during this writing effort.

Very special thanks to Peter Bunus for help with model examples, some figures, MicroSoft Word formatting, and for many inspiring discussions. Without your help this project might have been too hard, especially considering the damage to my hands from too much typing on computer keyboards.

Many thanks to Hilding Elmqvist for sharing the vision about a declarative modeling language, for starting off the Modelica design effort by inviting people to form a design group, for serving as the first chairman of Modelica Assocation, and for enthusiasm and many design contributions including pushing for a unified class concept. Also thanks for inspiration regarding presentation material including finding historical examples of equations.

Many thanks to Martin Otter for serving as the second chairman of the Modelica Association, for enthusiasm and energy, design and Modelica library contributions, as well as inspiration regarding presentation material.

Many thanks to Eva-Lena Lengquist Sandelin and Susanna Monemar for help with the exercises, for help with preparing certain appendices, and for preparing the DrModelica interactive notebook teaching material which makes the examples in this book more accessible for interactive learning and experimentation.

Thanks to Peter Aronsson, Bernhard Bachmann, Peter Beater, Jan Brugård, Dag Brück, Brian Elmegaard, Hilding Elmqvist, Vadim Engelson, Rüdiger Franke, Dag Fritzson, Torkel Glad, Pavel Grozman, Daniel Hedberg, Andreas Idebrant, Mats Jirstrand, Olof Johansson, David Landén, Emma Larsdotter Nilsson, Håkan Lundvall, Sven-Erik Mattsson, Iakov Nakhimovski, Hans Olsson, Adrian Pop, Per Sahlin, Levon Saldamli, Hubertus Tummescheit, and Hans-Jürg Wiesmann for constructive comments, and in some cases other help, on parts of the book, and to Peter Bunus and Dan Costello for help in making MicroSoft Word more cooperative.

Thanks to Hans Olsson and Dag Brück, who edited several versions of the Modelica Specification, and to Michael Tiller for sharing my interest in programming tools and demonstrating that it is indeed possible to write a Modelica book.

Thanks to Bodil Mattsson-Kihlström for handling many administrative chores at the Programming Environment Laboratory while I have been focusing on book writing, to Ulf Nässén for inspiration and encouragement, and to Uwe Assmann for encouragement and sharing common experiences on the hard task of writing books.

Thanks to all members of PELAB and employees of MathCore Engineering, who have given comments and feedback.

Finally, thanks to the staff at Vårdnäs Stiftgård, who have provided a pleasant atmosphere for important parts of this writing effort.

A final note: approximately 95 per cent of the running text in this book has been entered by voice using Dragon Naturally Speaking. This is usually slower than typing, but still quite useful for a person like me, who has acquired RSI (Repetitive Strain Injury) due to too much typing. Fortunately, I can still do limited typing and drawing, e.g., for corrections, examples, and figures. All Modelica examples are hand-typed, but often with the help of others. All figures except the curve diagrams are, of course, hand drawn.

Linköping, September 2003

Peter Fritzson

Contributions to Examples

Many people contributed to the original versions of some of the Modelica examples presented in this book. Most examples have been significantly revised compared to the originals. A number of individuals are acknowledged below with the risk of accidental omission due to oversight. If the original version of an example is from the Modelica Tutorial or the Modelica Specification on the Modelica Association web sites, the contributors are the members of the Modelica Association. In addition to the examples mentioned in this table, there are also numerous small example fragments from the Modelica Tutorial and Specification used in original or modified form in the text, which is indicated to some extent in the refererence section of each chapter.

Classes	*Individuals*
`VanDerPol` in Section 2.1.1	Andreas Karström
`SimpleCircuit` in Section 2.7.1	Members of the Modelica Association.
`PolynomialEvaluator` in Section 2.14.3	Members of the Modelica Association.
`LeastSquares` in Section 2.14.4	Mikael Adlers
`Diode` and `BouncingBall` in Section 2.15	Members of the Modelica Association.
`SimpleCircuit` expansion in Section 2.20.1	Martin Otter
`Rocket` in Section 3.5	Peter Bunus
`MoonLanding` in Section 3.5	Peter Bunus
`BoardExample` in Section 3.13.3	Members of the Modelica Association.
`LowPassFilter` in Section 4.2.9	Members of the Modelica Association.
`FrictionFunction`, `KindOfController` Sec 4.3.7	Members of the Modelica Association.
`Tank` in Section 4.4.5	Peter Bunus
`Oscillator`, `Mass`, `Rigid` in Section 5.4.4	Martin Otter
`RealInput`, `RealoutPut`, `MISO` in Section 5.5.2	Martin Otter
`MatrixGain`, `CrossProduct` in Section 5.7.4	Members of the Modelica Association.
`Environment` in Section 5.8.1	Members of the Modelica Association.
`CircuitBoard` in Section 5.8.2	Members of the Modelica Association.
`uniformGravity`, `pointGravity` in Section 5.8.3	Members of the Modelica Association.
`ParticleSystem` in Section 5.8.3	Members of the Modelica Association.
`PendulumImplicitL`, `readParameterData` in Section 8.4.1.4	Sven-Erik Mattsson, Hilding Elmqvist, Martin Otter, Hans Olsson
`ProcessControl1`, `ProcessControl2`, `ProcessControl3`, `ProcessControl4` in Section 8.4.2	Sven-Erik Mattsson, Hilding Elmqvist, Martin Otter, Hans Olsson
`HeatRectangle2D` in Section 8.5.1.4	Levon Saldamli
Material to Figure 8-9 on 2D heat flow using FEM.	Levon Saldamli
`FieldDomainOperators1D` in Section 8.5.4.	Hilding Elmqvist, Jonas Jonasson
`DifferentialOperators1D` in Section 8.5.4.	Jonas Jonasson, Hilding Elmqvist
`HeadDiffusion1D` in Section 8.5.4.	Jonas Jonasson, Hilding Elmqvist
`Diff22D` in Section 8.5.4.1	Hilding Elmqvist
`FourBar1` example in Section 8.6.1.	Martin Otter
`Orientation` in Section 8.6.1.	Martin Otter, Hilding Elmqvist, Sven-Erik Mattsson.
`FixedTranslation` in Section 8.6.2	Martin Otter, Hilding Elmqvist, Sven-Erik Mattsson.
Material to Figure 8-14 on cutting branches in virtual connection graph.	Martin Otter, Hilding Elmqvist, Sven-Erik Mattsson.

`findElement` in Section 9.2.7	Peter Aronsson
`FilterBlock1` in Section 9.2.10.1	Members of the Modelica Association.
`realToString` in Section 9.3.2.1	Members of the Modelica Association.
`eigen` in Section 9.3.2.3	Martin Otter
`findMatrixElement` in Section 9.3.2.5	Peter Aronsson
`Record2` in Section 9.3.2.6	Members of the Modelica Association.
`bilinearSampling` in Section 9.4.3	Members of the Modelica Association.
`MyTable, interpolateMyTable` in Section 9.4.6	Members of the Modelica Association.
`Mechanics` in Section 10.3.2.2	Members of the Modelica Association.
`Placement`, `Transformation` in Section 11.3.4	Members of the Modelica Association.
`Line`, `Polygon`, etc. in Section 11.3.5	Members of the Modelica Association.
`Resistor`, `FrictionFunction` in Section 11.3.6.2	Members of the Modelica Association.
`h0`, `h1`, `h2` in Section 11.5.1	Members of the Modelica Association.
`FlatTank` in Section 12.2.1.1	Peter Bunus
`TankPI`, `Tank`, `LiquidSource` in Section 12.2.3	Peter Bunus
`PIContinuousController` in Section 12.2.3	Peter Bunus
`TankPID`, `PIContinuousController` Section 12.2.4	Peter Bunus
`DC-Motor Servo` in Section 12.3	Mats Jirstrand
`WatchDogSystem` in Section 13.3.2.2	Peter Bunus
`CustomerGeneration` in Section 13.3.4.2	Peter Bunus
`ServerWithQueue` in Section 13.3.4.2	Peter Bunus
`BasicDEVSTwoPort` in Section 13.3.5	Peter Bunus
`SimpleDEVSServer` in Section 13.3.5	Peter Bunus
`Place`, `Transition` in Section 13.3.6.5	Hilding Elmqvist, Peter Bunus
`GameOfLife`, `nextGeneration` in Section 13.3.3.1	Peter Bunus
`PIdiscreteController` in Section 13.4.1	Peter Bunus
`TankHybridPI` in Section 13.4.1	Peter Bunus
`SimpleElastoBacklash` in Section 13.4.2	Peter Bunus
`DCMotorCircuitBacklash` in Section 13.4.2	Peter Bunus
`ElastoBacklash` in Section 13.4.2	Martin Otter
`Philosophers`, `DiningTable` in Section 13.5.1	Håkan Lundvall
`BasicVolume` in Section 15.2.2	Mats Jirstrand
`BasicVolume` in Section 15.2.3	Mats Jirstrand
`BasicVolume` in Section 15.2.4	Mats Jirstrand
`BasicVolume` in Section 15.2.4.2	Johan Gunnarsson
`IdealGas` in Section 15.2.5.2	Mats Jirstrand, Hubertus Tummescheit
`PressureEnthalpySource` in Section 15.2.5.3	Mats Jirstrand
`SimpleValveFlow` in Section 15.2.5.4	Mats Jirstrand
`ValveFlow` in Section 15.2.5.5	Mats Jirstrand
`ControlledValveFlow` in Section 15.2.5.6	Mats Jirstrand
`CtrlFlowSystem` in Section 15.2.5.6	Mats Jirstrand
`PneumaticCylinderVolume` in Section 15.2.5.7	Hubertus Tummescheit
`PneumaticCylinderVolume` in Section 15.2.5.8	Hubertus Tummescheit
`RoomWithFan` in Section 15.2.6	Hubertus Tummescheit
`RoomInEnvironment` in Section 15.2.6	Hubertus Tummescheit
`HydrogenIodide` in Section 15.3.1	Emma Larsdotter Nilsson
`LotkaVolterra` in Section 15.4.1	Emma Larsdotter Nilsson
`WillowForest` in Section 15.4.2	Emma Larsdotter Nilsson

`TCPSender`, `Loss_link_queue` in Section 15.6.3	Daniel Färnquist et. al., Peter Bunus
`Router21`, `TCPSackvsTCPWestWood` in Section 15.6	Daniel Färnquist et. al., Peter Bunus
`LinearActuator` in Section 15.7	Mats Jirstrand, Jan Brugård
`WeakAxis` in Section 15.8	Mats Jirstrand, Jan Brugård
`WaveEquationSample` in Section 15.9	Jan Brugård, Mats Jirstrand
`FreeFlyingBody` in Section 15.10.2	Vadim Engelson
`doublePendulumNoGeometry` in Section 15.10.6	Vadim Engelson
`doublePendulumCylinders` in Section 15.10.5.2	Vadim Engelson
`PendulumLoop2D` in Section 15.10.6	Vadim Engelson
`ExtForcePendulum` in Section 15.10.8.1	Vadim Engelson
`PendulumRotationalSteering` in Section 15.10.8.3	Vadim Engelson
`PendulumWithPDController` in Section 15.10.8.4	Vadim Engelson
`TripleSprings` in Section `TripleSprings` 5.4.3.2	Martin Otter
`EngineV6` in Section 15.10.10	Martin Otter
`Generator` in Section 17.1.6	Peter Bunus
Material to Figure 17-4, Figure 17-5, and Figure 17-6	Bernhard Bachmann
`EntertainmentUnit` in Section 18.3.1.1	Karin Lunde
`LayerConstraints` in Section 18.3.1.1	Members of Modelica Association
`CdChangerBase, CdChanger` in Section 18.3.1.2	Hans Olsson

Contributors to the Modelica Standard Library, Versions 1.0 to 2.1

Person	*Affiliation*
Peter Beater	University of Paderborn, Germany
Christoph Clauß	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Martin Otter	German Aerospace Center, Oberpfaffenhofen, Germanyr
André Schneider	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Nikolaus Schürmann	German Aerospace Center, Oberpfaffenhofen, Germany
Christian Schweiger	German Aerospace Center, Oberpfaffenhofen, Germany
Michael Tiller	Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit	Lund Institute of Technology, Sweden

Contributors to the Modelica Language, Version 2.1

Person	*Affiliation*
Mikael Adlers	MathCore, Linköping, Sweden
Peter Aronsson	Linköping University, Sweden
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Peter Bunus	Linköping University, Sweden
Jonas Eborn	United Technologies Research Center, Hartford, U.S.A.
Hilding Elmqvist	Dynasim, Lund, Sweden
Rüdiger Franke	ABB Corporate Research, Ladenburg, Germany
Peter Fritzson	Linköping University, Sweden
Anton Haumer	Tech. Consult. & Electrical Eng., St.Andrae-Woerdern, Austria
Olof Johansson	Linköping University, Sweden
Karin Lunde	R.O.S.E. Informatik GmbH, Heidenheim, Germany
Sven Erik Mattsson	Dynasim, Lund, Sweden
Hans Olsson	Dynasim, Lund, Sweden

Martin Otter	German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli	Linköping University, Sweden
Christian Schweiger	German Aerospace Center, Oberpfaffenhofen, Germany
Michael Tiller	Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit	United Technologies Research Center, Hartford, U.S.A
Hans-Jürg Wiesmann	ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

Contributors to the Modelica Language, Version 2.0

Person	*Affiliation*
Peter Aronsson	Linköping University, Sweden
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Peter Beater	University of Paderborn, Germany
Dag Brück	Dynasim, Lund, Sweden
Peter Bunus	Linköping University, Sweden
Hilding Elmqvist	Dynasim, Lund, Sweden
Vadim Engelson	Linköping University, Sweden
Rüdiger Franke	ABB Corporate Research, Ladenburg
Peter Fritzson	Linköping University, Sweden
Pavel Grozman	Equa, Stockholm, Sweden
Johan Gunnarsson	MathCore, Linköping, Sweden
Mats Jirstrand	MathCore, Linköping, Sweden
Sven Erik Mattsson	Dynasim, Lund, Sweden
Hans Olsson	Dynasim, Lund, Sweden
Martin Otter	German Aerospace Center, Oberpfaffenhofen, Germany
Levon Saldamli	Linköping University, Sweden
Michael Tiller	Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit	Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann	ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

Contributors to the Modelica Language, Version 1.4

Person	*Affiliation*
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Dag Brück	Dynasim, Lund, Sweden
Peter Bunus	Linköping University, Sweden
Hilding Elmqvist	Dynasim, Lund, Sweden
Vadim Engelson	Linköping University, Sweden
Jorge Ferreira	University of Aveiro, Portugal
Peter Fritzson	Linköping University, Sweden
Pavel Grozman	Equa, Stockholm, Sweden
Johan Gunnarsson	MathCore, Linköping, Sweden
Mats Jirstrand	MathCore, Linköping, Sweden
Clemens Klein-Robbenhaar	Germany
Pontus Lidman	MathCore, Linköping, Sweden
Sven Erik Mattsson	Dynasim, Lund, Sweden
Hans Olsson	Dynasim, Lund, Sweden
Martin Otter	German Aerospace Center, Oberpfaffenhofen, Germany

Tommy Persson	Linköping University, Sweden
Levon Saldamli	Linköping University, Sweden
André Schneider	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Michael Tiller	Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit	Lund Institute of Technology, Sweden
Hans-Jürg Wiesmann	ABB Switzerland Ltd.,Corporate Research, Baden, Switzerland

Contributors to the Modelica Language, up to Version 1.3

Person	*Affiliation*
Bernhard Bachmann	University of Applied Sciences, Bielefeld, Germany
Francois Boudaud	Gaz de France, Paris, France
Jan Broenink	University of Twente, Enschede, the Netherlands
Dag Brück	Dynasim, Lund, Sweden
Thilo Ernst	GMD FIRST, Berlin, Germany
Hilding Elmqvist[1]	Dynasim, Lund, Sweden
Rüdiger Franke	ABB Network Partner Ltd. Baden, Switzerland
Peter Fritzson	Linköping University, Sweden
Pavel Grozman	Equa, Stockholm, Sweden
Kaj Juslin	VTT, Espoo, Finland
David Kågedal	Linköping University, Sweden
Mattias Klose	Technical University of Berlin, Germany
N. Loubere	Gaz de France, Paris, France
Sven Erik Mattsson	Dynasim, Lund, Sweden
P. J. Mosterman	German Aerospace Center, Oberpfaffenhofen, Germany
Henrik Nilsson	Linköping University, Sweden
Hans Olsson	Dynasim, Lund, Sweden
Martin Otter	German Aerospace Center, Oberpfaffenhofen, Germany
Per Sahlin	Bris Data AB, Stockholm, Sweden
André Schneider	Fraunhofer Institute for Integrated Circuits, Dresden, Germany
Michael Tiller	Ford Motor Company, Dearborn, MI, U.S.A.
Hubertus Tummescheit	Lund Institute of Technology, Sweden
Hans Vangheluwe	University of Gent, Belgium

Modelica Association Member Companies and Organizations

Company or Organization

Dynasim AB, Lund, Sweden

Equa AB, Stockholm, Sweden

German Aerospace Center, Oberpfaffenhofen, Germany

Linköping University, the Programming Environment Laboratory, Linköping, Sweden

MathCore Engineering AB, Linköping, Sweden

[1] The Modelica 1.0 design document was edited by Hilding Elmqvist.

Funding Contributions

Linköping University, the Swedish Strategic Research Foundation (SSF) in the ECSEL graduate school, and MathCore AB have supported me for part of this work. Additionally, results in the form of reports and papers produced in a number of research projects at my research group PELAB, the Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, have been used as raw material in writing parts of this book. Support for these research projects have been given by: the Swedish Business Development Agency (NUTEK) in the Modelica Tools project, the Wallenberg foundation in the WITAS project, the Swedish Agency for Innovation Systems (VINNOVA) in the VISP (Virtuell Integrerad Simuleringsstödd Produktframtagning) project, the European Union in the REALSIM (Real-Time Simulation for Design of Multi-physics Systems) project, SSF in the VISIMOD (Visualization, Identification, Simulation, and Modeling) project, and SKF AB.

Table of Contents

Chapter 1

Introduction to Modeling and Simulation

It is often said that computers are revolutionizing science and engineering. By using computers we are able to construct complex engineering designs such as space shuttles. We are able to compute the properties of the universe as it was fractions of a second after the big bang. Our ambitions are ever-increasing. We want to create even more complex designs such as better spaceships, cars, medicines, computerized cellular phone systems, etc. We want to understand deeper aspects of nature. These are just a few examples of computer-supported modeling and simulation. More powerful tools and concepts are needed to help us handle this increasing complexity, which is precisely what this book is about.

This text presents an object-oriented component-based approach to computer-supported mathematical modeling and simulation through the powerful Modelica language and its associated technology. Modelica can be viewed as an almost-universal approach to high-level computational modeling and simulation, by being able to represent a range of application areas and providing general notation as well as powerful abstractions and efficient implementations. The introductory part of this book consisting of the first two chapters gives a quick overview of the two main topics of this text:

- Modeling and simulation.
- The Modelica language.

The two subjects are presented together since they belong together. Throughout the text Modelica is used as a vehicle for explaining different aspects of modeling and simulation. Conversely, a number of concepts in the Modelica language are presented by modeling and simulation examples. This first chapter introduces basic concepts such as *system*, *model*, and *simulation*. The second chapter gives a quick tour of the Modelica language as well as a number of examples, interspersed with presentations of topics such as object-oriented mathematical modeling, declarative formalisms, methods for compilation of equation-based models, etc.

Subsequent chapters contain detailed presentations of object-oriented modeling principles and specific Modelica features, introductions of modeling methodology for continuous, discrete, and hybrid systems, as well as a thorough overview of a number of currently available Modelica model libraries for a range of application domains. Finally, in the last chapter, a few of the currently available Modelica environments are presented.

1.1 Systems and Experiments

What is a system? We have already mentioned some systems such as the universe, a space shuttle, etc. A system can be almost anything. A system can contain subsystems which are themselves systems. A possible definition of system might be:

- A system is an object or collection of objects whose properties we want to study.

Our wish to study selected properties of objects is central in this definition. The “study” aspect is fine despite the fact that it is subjective. The selection and definition of what constitutes a system is somewhat arbitrary and must be guided by what the system is to be used for.

What reasons can there be to study a system? There are many answers to this question but we can discern two major motivations:

- Study a system to understand it in order to build it. This is the engineering point of view.
- Satisfy human curiosity, e.g. to understand more about nature—the natural science viewpoint.

1.1.1 Natural and Artificial Systems

A system according to our previous definition can occur naturally, e.g. the universe, it can be artificial such as a space shuttle, or a mix of both. For example, the house in Figure 1-1 with solar-heated tap warm water is an artificial system, i.e., manufactured by humans. If we also include the sun and clouds in the system it becomes a combination of natural and artificial components.

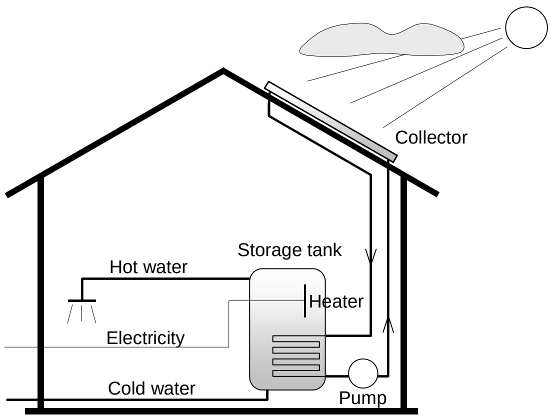


Figure 1-1. A system: a house with solar-heated tap warm water, together with clouds and sunshine.

Even if a system occurs naturally its definition is always highly selective. This is made very apparent in the following quote from Ross Ashby [Ashby-56]:

> At this point, we must be clear about how a *system* is to be defined. Our first impulse is to point at the *pendulum* and to say “the system is that thing there.” This method, however, has a fundamental disadvantage: every material object contains no less than an infinity of variables, and therefore, of possible systems. The real pendulum, for instance, has not only length and position; it has also mass, temperature, electric conductivity, crystalline structure, chemical impurities, some radioactivity, velocity, reflecting power, tensile strength, a surface film of moisture, bacterial contamination, an optical absorption, elasticity, shape, specific gravity, and so on and on. Any suggestion that we should study all the facts is unrealistic, and actually the attempt is never made. What is necessary is that we should pick out and study the facts that are relevant to some main interest that is already given.

Even if the system is completely artificial, such as the cellular phone system depicted in Figure 1-2, we must be highly selective in its definition depending on what aspects we want to study for the moment.

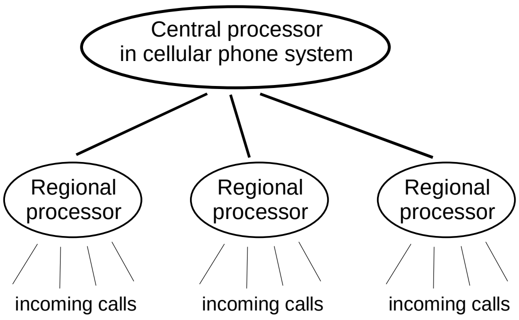


Figure 1-2. A cellular phone system containing a central processor and regional processors to handle incoming calls.

An important property of systems is that they should be *observable*. Some systems, but not large natural systems like the universe, are also *controllable* in the sense that we can influence their behavior through inputs, i.e.:

- The *inputs* of a system are variables of the environment that influence the behavior of the system. These inputs may or may not be controllable by us.
- The *outputs* of a system are variables that are determined by the system and may influence the surrounding environment.

In many systems the same variables act as *both inputs and outputs*. We talk about *acausal* behavior if the relationships or influences between variables do not have a causal direction, which is the case for relationships described by equations. For example, in a mechanical system the forces from the environment influence the displacement of an object, but on the other hand the displacement of the object influences the forces between the object and environment. What is input and what is output in this case is primarily a choice by the observer, guided by what is interesting to study, rather than a property of the system itself.

1.1.2 Experiments

Observability is essential in order to study a system according to our definition of system. We must at least be able to observe some outputs of a system. We can learn even more if it is possible to exercise a system by controlling its inputs. This process is called *experimentation*, i.e.:

- An *experiment* is the process of extracting information from a system by exercising its inputs.

To perform an experiment on a system it must be both controllable and observable. We apply a set of external conditions to the accessible inputs and observe the reaction of the system by measuring the accessible outputs.

One of the disadvantages of the experimental method is that for a large number of systems many inputs are not accessible and controllable. These systems are under the influence of inaccessible inputs, sometimes called *disturbance inputs*. Likewise, it is often the case that many really useful possible outputs are not accessible for measurements; these are sometimes called *internal states* of the system. There are also a number of practical problems associated with performing an experiment, e.g.:

- The experiment might be too *expensive*: investigating ship durability by building ships and letting them collide is a very expensive method of gaining information.
- The experiment might be too *dangerous*: training nuclear plant operators in handling dangerous situations by letting the nuclear reactor enter hazardous states is not advisable.
- The *system* needed for the experiment might *not yet exist*. This is typical of systems to be designed or manufactured.

The shortcomings of the experimental method lead us over to the model concept. If we make a model of a system, this model can be investigated and may answer many questions regarding the real system if the model is realistic enough.

1.2 The Model Concept

Given the previous definitions of system and experiment, we can now attempt to define the notion of model:

- A *model* of a system is anything an “experiment” can be applied to in order to answer questions about that *system*.

This implies that a model can be used to answer questions about a system *without* doing experiments on the *real* system. Instead we perform a kind of simplified “experiments” on the model, which in turn can be regarded as a kind of simplified system that reflects properties of the real system. In the simplest case a model can just be a piece of information that is used to answer questions about the system.

Given this definition, any model also qualifies as a system. Models, just like systems, are hierarchical in nature. We can cut out a piece of a model, which becomes a new model that is valid for a subset of the experiments for which the original model is valid. A model is always related to the system it models and the experiments it can be subject to. A statement such as “a model of a system is invalid” is meaningless without mentioning the associated system and the experiment. A model of a system might be valid for one experiment on the model and invalid for another. The term model *validation,* see Section 1.5.3on page 10, always refers to an experiment or a class of experiment to be performed.

We talk about different kinds of models depending on how the model is represented:

- *Mental* model—a statement like “a person is reliable” helps us answer questions about that person’s behavior in various situations.
- *Verbal* model—this kind of model is expressed in words. For example, the sentence “More accidents will occur if the speed limit is increased” is an example of a verbal model. Expert systems is a technology for formalizing verbal models.
- *Physical* model—this is a physical object that mimics some properties of a real system, to help us answer questions about that system. For example, during design of artifacts such as buildings, airplanes, etc., it is common to construct small physical models with same shape and appearance as the real objects to be studied, e.g. with respect to their aerodynamic properties and aesthetics.
- *Mathematical* model—a description of a system where the relationships between variables of the system are expressed in mathematical form. Variables can be measurable quantities such as size, length, weight, temperature, unemployment level, information flow, bit rate, etc. Most laws of nature are mathematical models in this sense. For example, Ohm’s law describes the relationship between current and voltage for a resistor; Newton’s laws describe relationships between velocity, acceleration, mass, force, etc.

The kinds of models that we primarily deal with in this book are mathematical models represented in various ways, e.g. as equations, functions, computer programs, etc. Artifacts represented by mathematical models in a computer are often called *virtual prototypes*. The process of constructing and investigating such models is virtual prototyping. Sometimes the term *physical modeling* is used also for the process of building mathematical models of physical systems in the computer if the structuring and synthesis process is the same as when building real physical models.

Chapter 2

A Quick Tour of Modelica

Modelica is primarily a modeling language that allows specification of mathematical models of complex natural or man-made systems, e.g., for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is also an object-oriented equation-based programming language, oriented toward computational applications with high complexity requiring high performance. The four most important features of Modelica are:

- Modelica is primarily based on equations instead of assignment statements. This permits acausal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context.
- Modelica has multidomain modeling capability, meaning that model components corresponding to physical objects from several different domains such as, e.g., electrical, mechanical, thermodynamic, hydraulic, biological, and control applications can be described and connected.
- Modelica is an object-oriented language with a general class concept that unifies classes, generics—known as templates in C++ —and general subtyping into a single language construct. This facilitates reuse of components and evolution of models.
- Modelica has a strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

These are the main properties that make Modelica both powerful and easy to use, especially for modeling and simulation. We will start with a gentle introduction to Modelica from the very beginning.

2.1 Getting Started with Modelica

Modelica programs are built from classes, also called models. From a class definition, it is possible to create any number of objects that are known as instances of that class. Think of a class as a collection of blueprints and instructions used by a factory to create objects. In this case the Modelica compiler and run-time system is the factory.

A Modelica class contains elements, the main kind being variable declarations, and equation sections containing equations. Variables contain data belonging to instances of the class; they make up the data storage of the instance. The equations of a class specify the behavior of instances of that class.

There is a long tradition that the first sample program in any computer language is a trivial program printing the string `"Hello World"`. Since Modelica is an equation-based language, printing a string does not make much sense. Instead, our Hello World Modelica program solves a trivial *differential equation*:

$$\dot{x} = -a \cdot x \tag{2-1}$$

The variable `x` in this equation is a dynamic variable (here also a state variable) that can change value over time. The time derivative $\dot{x}$ is the time derivative of `x`, represented as `der(x)` in Modelica. Since all Modelica programs, usually called *models*, consist of class declarations, our `HelloWorld` program is declared as a class:

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = -a*x;
end HelloWorld;
```

Use your favorite text editor or Modelica programming environment to type in this Modelica code[4], or open the DrModelica electronic document containing all examples and exercises in this book. Then invoke the simulation command in your Modelica environment. This will compile the Modelica code to some intermediate code, usually C code, which in turn will be compiled to machine code and executed together with a numerical ordinary differential equation (ODE) solver or differential algebraic equation (DAE) solver to produce a solution for `x` as a function of time. The following command in the MathModelica or OpenModelica environments produces a solution between time 0 and time 2:

```
simulate5(HelloWorld,stopTime=2)
```

Since the solution for `x` is a function of time, it can be plotted by a plot command:

```
plot6(x)
```

(or the longer form `plot(x,xrange={0,2})` that specifies the x-axis), giving the curve in Figure 2-1:

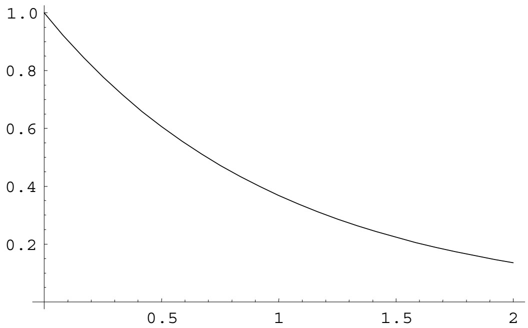


Figure 2-1. Plot of a simulation of the simple `HelloWorld` model.

Now we have a small Modelica model that does something, but what does it actually mean? The program contains a declaration of a class called `HelloWorld` with two variables and a single equation. The first attribute of the class is the variable `x`, which is initialized to a start value of 1 at the time when the simulation starts. All variables in Modelica have a `start` attribute with a default value which is normally set to 0. Having a different start value is accomplished by providing a so-called modifier

[4] There is a Modelica environment called MathModelica from MathCore (www.mathcore.com), Dymola from Dynasim (www.dynasim.se), and OpenModelica from Linköping University (www.ida.liu.se/labs/pelab/modelica).

[5] `simulate` is the MathModelica Modelica-style and OpenModelica command for simulation. The corresponding MathModelica Mathematica-style command for this example, would be `Simulate[HelloWorld, {t,0,2}]`, and in Dymola `simulateModel("HelloWorld", stopTime=2)`.

[6] `plot` is the MathModelica Modelica-style command for plotting simulation results. The corresponding MathModelica Mathematica-style and Dymola commands would be `PlotSimulation[x[t], {t,0,2}]`, and `plot({"x"})` respectively.

within parentheses after the variable name, i.e., a modification equation setting the start attribute to 1 and replacing the original default equation for the attribute.

The second attribute is the variable `a`, which is a constant that is initialized to 1 at the beginning of the simulation. Such a constant is prefixed by the keyword `parameter` in order to indicate that it is constant during simulation but is a model parameter that can be changed between simulations, e.g., through a command in the simulation environment. For example, we could rerun the simulation for a different value of `a`.

Also note that each variable has a type that precedes its name when the variable is declared. In this case both the variable `x` and the “variable” `a` have the type `Real`.

The single equation in this `HelloWorld` example specifies that the time derivative of `x` is equal to the constant `-a` times `x`. In Modelica the equal sign `=` always means equality, i.e., establishes an equation, and not an assignment as in many other languages. Time derivative of a variable is indicated by the pseudofunction `der( )`.

Our second example is only slightly more complicated, containing five rather simple equations (2-2):

$$
\begin{aligned}
m\dot{v}_x &= -\frac{x}{L}F \\
m\dot{v}_y &= -\frac{y}{L}F - mg \\
\dot{x} &= v_x \\
\dot{y} &= v_y \\
x^2 + y^2 &= L^2
\end{aligned}
\tag{2-2}
$$

This example is actually a mathematical model of a physical system, a planar pendulum, as depicted in Figure 2-2.

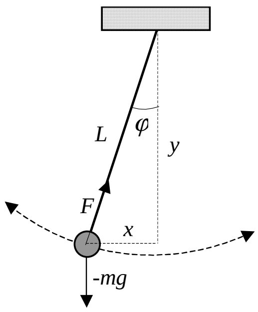


Figure 2-2. A planar pendulum.

The equations are Newton’s equations of motion for the pendulum mass under the influence of gravity, together with a geometric constraint, the 5th equation $x^2 + y^2 = L^2$, that specifies that its position (*x*,*y*) must be on a circle with radius *L*. The variables v_x and v_y are its velocities in the *x* and *y* directions respectively.

The interesting property of this model, however, is the fact that the 5th equation is of a different kind: a so-called *algebraic equation* only involving algebraic formulas of variables but no derivatives. The first four equations of this model are differential equations as in the `HelloWorld` example. Equation systems that contain both differential and algebraic equations are called *differential algebraic equation systems* (DAEs). A Modelica model of the pendulum appears below:

```
class Pendulum  "Planar Pendulum"
  constant  Real PI=3.141592653589793;
  parameter Real m=1, g=9.81, L=0.5;
  Real F;
  output    Real x(start=0.5),y(start=0);
```

```
  output    Real vx,vy;
equation
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-m*g;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end Pendulum;
```

We simulate the `Pendulum` model and plot the x-coordinate, shown in Figure 2-3:

```
simulate(Pendulum, stopTime=4)
plot(x);
```

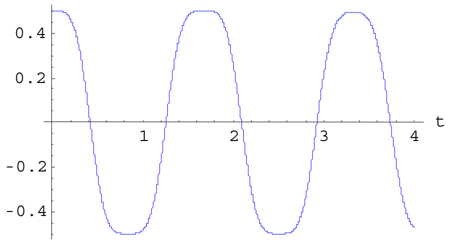


Figure 2-3. Plot of a simulation of the Pendulum DAE (differential algebraic equation) model.

You can also write down DAE equation systems without physical significance, with equations containing formulas selected more or less at random, as in the class `DAEexample` below:

```
class DAEexample
  Real x(start=0.9);
  Real y;
equation
  der(y) + (1+0.5*sin(y))*der(x) = sin(time);
  x-y = exp(-0.9*x)*cos(y);
end DAEexample;
```

This class contains one differential and one algebraic equation. Try to simulate and plot it yourself, to see if any reasonable curve appears!

Finally, an important observation regarding Modelica models:

- The number of *variables* must be equal to the number of *equations*!

This statement is true for the three models we have seen so far, and holds for all solvable Modelica models. By variables we mean something that can vary, i.e., not named constants and parameters to be described in Section 2.1.3, page 24.

2.1.1 Variables

This example shows a slightly more complicated model, which describes a Van der Pol[7] oscillator. Notice that here the keyword `model` is used instead of `class` with almost the same meaning.

```
model Van,DerPol   "Van der Pol oscillator model"
  Real x(start = 1)   "Descriptive string for x";  // x starts at 1
  Real y(start = 1)   "Descriptive string for y";  // y starts at 1
  parameter Real lambda = 0.3;
equation
```

[7] Balthazar van der Pol was a Dutch electrical engineer who initiated modern experimental dynamics in the laboratory during the 1920's and 1930's. Van der Pol investigated electrical circuits employing vacuum tubes and found that they have stable oscillations, now called limit cycles. The van der Pol oscillator is a model developed by him to describe the behavior of nonlinear vacuum tube circuits

```
    der(x) = y;                          // This is the first equation
    der(y) = -x + lambda*(1 - x*x)*y;    /* The 2nd differential equation */
  end VanDerPol;
```

This example contains declarations of two dynamic variables (here also state variables) `x` and `y`, both of type `Real` and having the start value 1 at the beginning of the simulation, which normally is at time 0. Then follows a declaration of the parameter constant lambda, which is a so-called model parameter.

The keyword `parameter` specifies that the variable is constant during a simulation run, but can have its value initialized before a run, or between runs. This means that parameter is a special kind of constant, which is implemented as a static variable that is initialized once and never changes its value during a specific execution. A `parameter` is a constant variable that makes it simple for a user to modify the behavior of a model, e.g., changing the parameter lambda which strongly influences the behavior of the Van der Pol oscillator. By contrast, a fixed Modelica constant declared with the prefix `constant` never changes and can be substituted by its value wherever it occurs.

Finally we present declarations of three dummy variables just to show variables of data types different from `Real`: the boolean variable `bb`, which has a default start value of `false` if nothing else is specified, the string variable `dummy` which is always equal to `"dummy string"`, and the integer variable `fooint` always equal to `0`.

```
Boolean bb;
String dummy  = "dummy string";
Integer fooint = 0;
```

Modelica has built-in “primitive” data types to support floating-point, integer, boolean, and string values. These primitive types contain data that Modelica understands directly, as opposed to class types defined by programmers. The type of each variable must be declared explicitly. The primitive data types of Modelica are:

`Boolean`	either `true` or `false`
`Integer`	corresponding to the C int data type, usually 32-bit two’s complement
`Real`	corresponding to the C double data type, usually 64-bit floating-point
`String`	string of text characters
`enumeration(...)`	enumeration type of enumeration literals

Finally, there is an equation section starting with the keyword `equation`, containing two mutually dependent equations that define the dynamics of the model.

To illustrate the behavior of the model, we give a command to simulate the Van der Pol oscillator during 25 seconds starting at time 0:

```
simulate(VanDerPol, stopTime=25)
```

A phase plane plot of the state variables for the Van der Pol oscillator model (Figure 2-4):

```
plotParametric(x,y, stopTime=25)
```

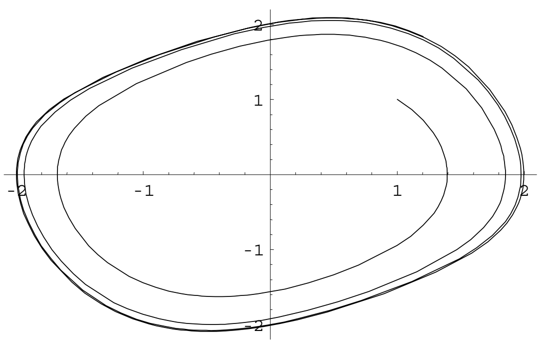


Figure 2-4. Parametric plot of a simulation of the Van der Pol oscillator model.

The names of variables, functions, classes, etc. are known as identifiers. There are two forms in Modelica. The most common form starts with a letter, followed by letters or digits, e.g. `x2`. The second form starts with a single-quote, followed by any characters, and terminated by a single-quote, e.g. `'2nd*3'`.

2.1.2 Comments

Arbitrary descriptive text, e.g., in English, inserted throughout a computer program, are comments to that code. Modelica has three styles of comments, all illustrated in the previous `VanDerPol` example.

Comments make it possible to write descriptive text together with the code, which makes a model easier to use for the user, or easier to understand for programmers who may read your code in the future. That programmer may very well be yourself, months or years later. You save yourself future effort by commenting your own code. Also, it is often the case that you find errors in your code when you write comments since when explaining your code you are forced to think about it once more.

The first kind of comment is a string within string quotes, e.g., “`a comment`”, optionally appearing after variable declarations, or at the beginning of class declarations. Those are “definition comments” that are processed to be used by the Modelica programming environment, e.g., to appear in menus or as help texts for the user. From a syntactic point of view they are not really comments since they are part of the language syntax. In the previous example such definition comments appear for the `VanDerPol` class and for the `x` and `y` variables.

The other two types of comments are ignored by the Modelica compiler, and are just present for the benefit of Modelica programmers. Text following `//` up to the end of the line is skipped by the compiler, as is text between `/* and the next */`. Hence the last type of comment can be used for large sections of text that occupies several lines.

Finally we should mention a construct called `annotation`, a kind of structured “comment” that can store information together with the code, described in Section 2.17.

2.1.3 Constants

Constant literals in Modelica can be integer values such as 4, 75, 3078; floating-point values like 3.14159, 0.5, 2.735E-10, 8.6835e+5; string values such as `"hello world"`, `"red"`; and enumeration values such as `Colors.red`, `Sizes.xlarge`.

Named constants are preferred by programmers for two reasons. One reason is that the name of the constant is a kind of documentation that can be used to describe what the particular value is used for. The other, perhaps even more important reason, is that a named constant is defined at a single place in the program. When the constant needs to be changed or corrected, it can be changed in only one place, simplifying program maintenance.

Named constants in Modelica are created by using one of the prefixes `constant` or `parameter` in declarations, and providing a declaration equation as part of the declaration. For example:

```
constant  Real    PI = 3.141592653589793;
constant  String  redcolor = "red";
constant  Integer one = 1;
parameter Real    mass = 22.5;
```

Parameter constants can be declared without a declaration equation since their value can be defined, e.g., by reading from a file, before simulation starts. For example:

```
parameter Real mass, gravity, length;
```

2.1.4 Default start Values

If a numeric variable lacks a specified definition value or `start` value in its declaration, it is usually initialized to zero at the start of the simulation. Boolean variables have `start` value `false`, and string variables the `start` value empty string `""` if nothing else is specified.

Exceptions to this rule are function *results* and *local* variables in functions, where the default initial value at function call is *undefined*. See also Section 8.4, page 250.

2.2 Object-Oriented Mathematical Modeling

Traditional object-oriented programming languages like Simula, C++, Java, and Smalltalk, as well as procedural languages such as Fortran or C, support programming with operations on stored data. The stored data of the program include variable values and object data. The number of objects often changes dynamically. The Smalltalk view of object-orientation emphasizes sending messages between (dynamically) created objects.

The Modelica view on object-orientation is different since the Modelica language emphasizes *structured* mathematical modeling. Object-orientation is viewed as a structuring concept that is used to handle the complexity of large system descriptions. A Modelica model is primarily a declarative mathematical description, which simplifies further analysis. Dynamic system properties are expressed in a declarative way through equations.

The concept of *declarative* programming is inspired by mathematics, where it is common to state or declare what *holds*, rather than giving a detailed stepwise *algorithm* on *how* to achieve the desired goal as is required when using procedural languages. This relieves the programmer from the burden of keeping track of such details. Furthermore, the code becomes more concise and easier to change without introducing errors.

Thus, the declarative Modelica view of object-orientation, from the point of view of object-oriented mathematical modeling, can be summarized as follows:

- Object-orientation is primarily used as a *structuring* concept, emphasizing the declarative structure and reuse of mathematical models. Our three ways of structuring are hierarchies, component-connections, and inheritance.
- Dynamic model properties are expressed in a declarative way through *equations*[8].
- An object is a collection of *instance* variables and equations that share a set of data.

However:

- Object-orientation in mathematical modeling is *not* viewed as dynamic message passing.

The declarative object-oriented way of describing systems and their behavior offered by Modelica is at a higher level of abstraction than the usual object-oriented programming since some implementation details can be omitted. For example, we do not need to write code to explicitly transport data between objects through assignment statements or message passing code. Such code is generated automatically by the Modelica compiler based on the given equations.

Just as in ordinary object-oriented languages, classes are blueprints for creating objects. Both variables and equations can be inherited between classes. Function definitions can also be inherited. However, specifying behavior is primarily done through equations instead of via methods. There are also facilities for stating algorithmic code including functions in Modelica, but this is an exception rather than the rule. See alsoChapter 3, page 73 and Chapter 4, page 111 for a discussion regarding object-oriented concepts.

---

[8] Algorithms are also allowed, but in a way that makes it possible to regard an algorithm section as a system of equations.

2.3 Classes and Instances

Modelica, like any object-oriented computer language, provides the notions of classes and objects, also called instances, as a tool for solving modeling and programming problems. Every object in Modelica has a class that defines its data and behavior. A class has three kinds of members:

- Data variables associated with a class and its instances. Variables represent results of computations caused by solving the equations of a class together with equations from other classes. During numeric solution of time-dependent problems, the variables stores results of the solution process at the current time instant.
- Equations specify the behavior of a class. The way in which the equations interact with equations from other classes determines the solution process, i.e., program execution.
- Classes can be members of other classes.

Here is the declaration of a simple class that might represent a point in a three-dimensional space:

```
class Point "Point in a three-dimensional space"
 public
  Real x;
  Real y, z;
end Point;
```

The `Point` class has three variables representing the `x`, `y`, and `z` coordinates of a point and has no equations. A class declaration like this one is like a blueprint that defines how instances created from that class look like, as well as instructions in the form of equations that define the behavior of those objects. Members of a class may be accessed using dot (`.`) notation. For example, regarding an instance `myPoint` of the `Point` class, we can access the `x` variable by writing `myPoint.x`.

Members of a class can have two levels of visibility. The `public` declaration of `x`, `y`, and `z`, which is default if nothing else is specified, means that any code with access to a `Point` instance can refer to those values. The other possible level of visibility, specified by the keyword `protected`, means that only code inside the class as well as code in classes that inherit this class, are allowed access.

Note that an occurrence of one of the keywords `public` or `protected` means that all member declarations following that keyword assume the corresponding visibility until another occurrence of one of those keywords, or the end of the class containing the member declarations has been reached.

2.3.1 Creating Instances

In Modelica, objects are created implicitly just by declaring instances of classes. This is in contrast to object-oriented languages like Java or C++, where object creation is specified using the new keyword. For example, to create three instances of our `Point` class we just declare three variables of type `Point` in a class, here `Triangle`:

```
class Triangle
  Point  point1;
  Point  point2;
  Point  point3;
end Triangle;
```

There is one remaining problem, however. In what context should `Triangle` be instantiated, and when should it just be interpreted as a library class not to be instantiated until actually used?

This problem is solved by regarding the class at the *top* of the instantiation hierarchy in the Modelica program to be executed as a kind of “main” class that is always implicitly instantiated, implying that its variables are instantiated, and that the variables of those variables are instantiated, etc. Therefore, to instantiate `Triangle`, either make the class `Triangle` the “top” class or declare an instance of

`Triangle` in the "main" class. In the following example, both the class `Triangle` and the class `Foo1` are instantiated.

```
class Foo1
  ...
end Foo1;

class Foo2
  ...
end Foo2;
...

class Triangle
  Point  point1;
  Point  point2;
  Point  point3;
end Triangle;

class Main
  Triangle  pts;
  foo1      f1;
end Main;
```

The variables of Modelica classes are instantiated per object. This means that a variable in one object is distinct from the variable with the same name in every other object instantiated from that class. Many object-oriented languages allow class variables. Such variables are specific to a class as opposed to instances of the class, and are shared among all objects of that class. The notion of class variables is not yet available in Modelica.

2.3.2 Initialization

Another problem is initialization of variables. As mentioned previously in Section 2.1.4, page 25, if nothing else is specified, the default start value of all numerical variables is zero, apart from function results and local variables where the initial value at call time is unspecified. Other start values can be specified by setting the `start` attribute of instance variables. Note that the start value only gives a suggestion for initial value—the solver may choose a different value unless the `fixed` attribute is true for that variable. Below a start value is specified in the example class `Triangle`:

```
class Triangle
  Point  point1(start={1,2,3});
  Point  point2;
  Point  point3;
end Triangle;
```

Alternatively, the start value of `point1` can be specified when instantiating `Triangle` as below:

```
class Main
  Triangle  pts(point1.start={1,2,3});
  foo1      f1;
end Main;
```

A more general way of initializing a set of variables according to some constraints is to specify an equation system to be solved in order to obtain the initial values of these variables. This method is supported in Modelica through the `initial equation` construct.

An example of a continuous-time controller initialized in steady-state, i.e., when derivatives should be zero, is given below:

```
model Controller
  Real y;
equation
```

```
    der(y) = a*y + b*u;
  initial equation
    der(y)=0;
  end Controller;
```

This has the following solution at initialization:

```
  der(y) = 0;
  y = -(b/a)*u;
```

For more information, see Section 8.4, page 250.

2.3.3 Restricted Classes

The class concept is fundamental to Modelica, and is used for a number of different purposes. Almost anything in Modelica is a class. However, in order to make Modelica code easier to read and maintain, special keywords have been introduced for specific uses of the class concept. The keywords `model`, `connector`, `record`, `block`, and `type` can be used instead of `class` under appropriate conditions. For example, a `record` is a class used to declare a record data structure and may not contain equations.

```
record Person
  Real   age;
  String name;
end Person;
```

A `block` is a class with fixed causality, which means that for each member variable of the class it is specified whether it has input or output causality. Thus, each variable in a `block` class interface must be declared with a causality prefix keyword of either `input` or `output`.

A `connector` class is used to declare the structure of “ports” or interface points of a component and may not contain equations. A `type` is a class that can be an alias or an extension to a predefined type, record, or array. For example:

```
type vector3D = Real[3];
```

Since restricted classes are just specialized versions of the general class concept, these keywords can be replaced by the `class` keyword for a valid Modelica model without changing the model behavior.

The idea of restricted classes is beneficial since the user does not have to learn several different concepts, except for one: the *class concept*. The notion of restricted classes gives the user a chance to express more precisely what a class is intended for, and requires the Modelica compiler to check that these usage constraints are actually fulfilled. Fortunately the notion is quite uniform since all basic properties of a class, such as the syntax and semantics of definition, instantiation, inheritance, and generic properties, are identical for all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified because only the syntax and semantics of the class concept have to be implemented along with some additional checks on restricted classes.

The `package` and `function` concepts in Modelica have much in common with the class concept but are not really restricted classes since these concepts carry additional special semantics of their own.

See also Section 3.8, page 81, regarding restricted classes.

2.3.4 Reuse of Modified Classes

The class concept is the key to reuse of modeling knowledge in Modelica. Provisions for expressing adaptations or modifications of classes through so-called modifiers in Modelica make reuse easier. For example, assume that we would like to connect two filter models with different time constants in series.

Instead of creating two separate filter classes, it is better to define a common filter class and create two appropriately modified instances of this class, which are connected. An example of connecting two modified low-pass filters is shown after the example low-pass filter class below:

```
model LowPassFilter
  parameter Real T=1  "Time constant of filter";
  Real u, y(start=1);
equation
  T*der(y) + y = u;
end LowPassFilter;
```

The model class can be used to create two instances of the filter with different time constants and "connecting" them together by the equation `F2.u = F1.y` as follows:

```
model FiltersInSeries
  LowPassFilter F1(T=2), F2(T=3);
equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltersInSeries;
```

Here we have used modifiers, i.e., attribute equations such as `T=2` and `T=3`, to modify the time constant of the low-pass filter when creating the instances `F1` and `F2`. The independent time variable is denoted `time`. If the `FiltersInSeries` model is used to declare variables at a higher hierarchical level, e.g., `F12`, the time constants can still be adapted by using hierarchical modification, as for `F1` and `F2` below:

```
model ModifiedFiltersInSeries
  FiltersInSeries F12(F1(T=6), F2.T=11);
end ModifiedFiltersInSeries;
```

See also Chapter 4, page 111.

2.3.5 Built-in Classes

The built-in type classes of Modelica correspond to the primitive types `Real`, `Integer`, `Boolean`, `String`, and `enumeration`(...), and have most of the properties of a class, e.g., can be inherited, modified, etc. Only the value attribute can be changed at run-time, and is accessed through the variable name itself, and not through dot notation, i.e., use `x` and not `x.value` to access the value. Other attributes are accessed through dot notation.

For example, a `Real` variable has a set of default attributes such as unit of measure, initial value, minimum and maximum value. These default attributes can be changed when declaring a new class, for example:

```
class Voltage = Real(unit= "V", min=-220.0, max=220.0);
```

See also Section 3.9, page 84.

2.4 Inheritance

One of the major benefits of object-orientation is the ability to extend the behavior and properties of an existing class. The original class, known as the *superclass* or *base class*, is extended to create a more specialized version of that class, known as the *subclass* or *derived class*. In this process, the behavior and properties of the original class in the form of variable declarations, equations, and other contents are reused, or inherited, by the subclass.

Let us regard an example of extending a simple Modelica class, e.g., the class `Point` introduced previously. First we introduce two classes named `ColorData` and `Color`, where `Color` inherits the data variables to represent the color from class `ColorData` and adds an equation as a constraint. The new class `ColoredPoint` inherits from multiple classes, i.e., uses multiple inheritance, to get the position variables from class `Point`, and the color variables together with the equation from class `Color`.

```
record ColorData
  Real red;
  Real blue;
  Real green;
end ColorData;

class Color
  extends ColorData;
equation
  red + blue + green = 1;
end Color;

class Point
 public
  Real x;
  Real y, z;
end Point;

class ColoredPoint
  extends Point;
  extends Color;
end ColoredPoint;
```

See also Chapter 4, page 111, regarding inheritance and reuse.

2.5 Generic Classes

In many situations it is advantageous to be able to express generic patterns for models or programs. Instead of writing many similar pieces of code with essentially the same structure, a substantial amount of coding and software maintenance can be avoided by directly expressing the general structure of the problem and providing the special cases as *parameter* values.

Such generic constructs are available in several programming languages, e.g., templates in C++, generics in Ada, and type parameters in functional languages such as Haskell or Standard ML. In Modelica the class construct is sufficiently general to handle generic modeling and programming in addition to the usual class functionality.

There are essentially two cases of generic class parameterization in Modelica: *class parameters* can either be *instance parameters*, i.e., have instance declarations (components) as values, or be *type parameters*, i.e., have types as values. Note that by class parameters in this context we do not usually mean model parameters prefixed by the keyword `parameter`, even though such “variables” are also a kind of class parameter. Instead we mean *formal parameters to the class*. Such formal parameters are prefixed by the keyword `replaceable`. The special case of replaceable local functions is roughly equivalent to virtual methods in some object-oriented programming languages.

See also Section 4.4, page 133.

2.5.1 Class Parameters Being Instances

First we present the case when class parameters are variables, i.e., declarations of instances, often called components. The class `C` in the example below has three class parameters *marked* by the keyword `replaceable`. These class parameters, which are components (variables) of class `C`, are declared as having the (default) types `GreenClass`, `YellowClass`, and `GreenClass` respectively. There is also a red object declaration which is not replaceable and therefore not a class parameter (Figure 2-5).

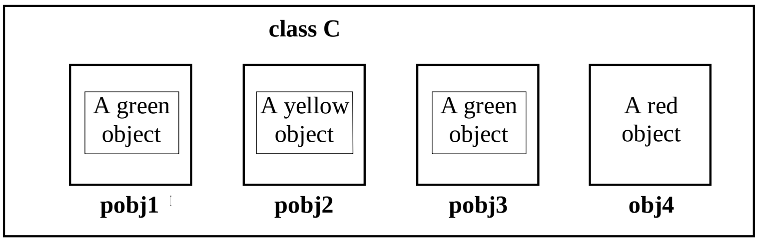


Figure 2-5. Three class parameters `pobj1`, `pobj2`, and `pobj3` that are instances (variables) of `class C`. These are essentially slots that can contain objects of different colors.

Here is the `class C` with its three class parameters `pobj1`, `pobj2`, and `pobj3` and a variable `obj4` that is not a class parameter:

```
class C
  replaceable GreenClass  pobj1(p1=5);
  replaceable YellowClass pobj2;
  replaceable GreenClass  pobj3;
  RedClass    obj4;
equation
  ...
end C;
```

Now a class `C2` is defined by providing two declarations of `pobj1` and `pobj2` as actual arguments to class `C`, being `red` and `green` respectively, instead of the defaults `green` and `yellow`. The keyword `redeclare` must precede an actual argument to a class formal parameter to allow changing its type. The requirement to use a keyword for a redeclaration in Modelica has been introduced in order to avoid accidentally changing the type of an object through a standard modifier.

In general, the type of a class component cannot be changed if it is not declared as `replaceable` and a redeclaration is provided. A variable in a redeclaration can replace the original variable if it has a type that is a subtype of the original type or its type constraint. It is also possible to declare type constraints (not shown here) on the substituted classes.

```
class C2 = C(redeclare RedClass pobj1, redeclare GreenClass pobj2);
```

Such a class `C2` obtained through redeclaration of `pobj1` and `pobj2` is of course equivalent to directly defining `C2` without reusing class `C`, as below.

```
class C2
  RedClass   pobj1(p1=5);
  GreenClass pobj2;
  GreenClass pobj3;
  RedClass   obj4;
equation
  ...
end C2;
```

2.5.2 Class Parameters being Types

A class parameter can also be a type, which is useful for changing the type of many objects. For example, by providing a type parameter `ColoredClass` in class `C` below, it is easy to change the color of all objects of type `ColoredClass`.

```
class C
  replaceable class ColoredClass = GreenClass;
```

```
  ColoredClass             obj1(p1=5);
  replaceable YellowClass obj2;
  ColoredClass             obj3;
  RedClass                 obj4;
equation
  ...
end C;
```

Figure 2-6 depicts how the type value of the `ColoredClass` class parameter is propagated to the member object declarations `obj1` and `obj3`.

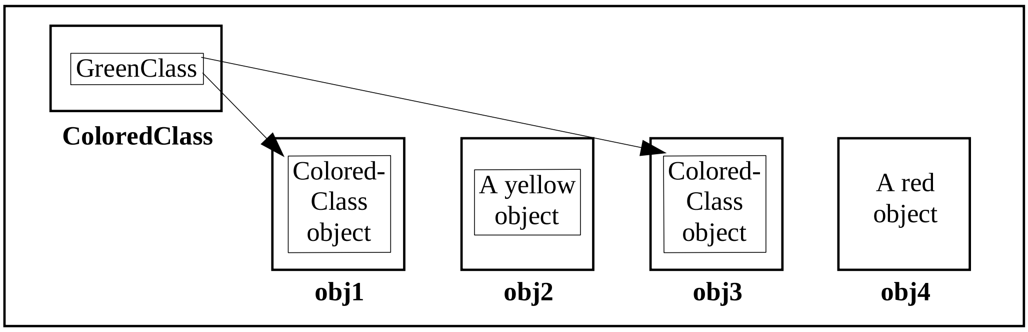


Figure 2-6. The class parameter `ColoredClass` is a type parameter that is propagated to the two member instance declarations of obj1 and obj3.

We create a class `C2` by giving the type parameter `ColoredClass` of class `C` the value `BlueClass`.

```
class C2 = C(redeclare class ColoredClass = BlueClass);
```

This is equivalent to the following definition of `C2`:

```
class C2
  BlueClass   obj1(p1=5);
  YellowClass obj2;
  BlueClass   obj3;
  RedClass    obj4;
equation
  ...
end C2;
```

2.6 Equations

As we already stated, Modelica is primarily an equation-based language in contrast to ordinary programming languages, where assignment statements proliferate. Equations are more flexible than assignments since they do not prescribe a certain data flow direction or execution order. This is the key to the physical modeling capabilities and increased reuse potential of Modelica classes.

Thinking in equations is a bit unusual for most programmers. In Modelica the following holds:

- Assignment statements in conventional languages are usually represented as equations in Modelica.
- Attribute assignments are represented as equations.
- Connections between objects generate equations.

Equations are more powerful than assignment statements. For example, consider a resistor equation where the resistance `R` multiplied by the current `i` is equal to the voltage `v`:

```
R*i = v;
```

This equation can be used in three ways corresponding to three possible assignment statements: computing the current from the voltage and the resistance, computing the voltage from the resistance and the current, or computing the resistance from the voltage and the current. This is expressed in the following three assignment statements:

```
i := v/R;
v := R*i;
R := v/i;
```

Equations in Modelica can be informally classified into four different groups depending on the syntactic context in which they occur:

- *Normal equations* occurring in equation sections, including the connect equation, which is a special form of equation.
- *Declaration equations*, which are part of variable or constant declarations.
- *Modification equations*, which are commonly used to modify attributes.
- *Initial equations*, specified in initial equation sections or as start attribute equations. These equations are used to solve the initialization problem at startup time.

As we already have seen in several examples, normal equations appear in equation sections started by the keyword `equation` and terminated by some other allowed keyword:

```
equation
  ...
  <equations>
  ...
<some other allowed keyword>
```

The above resistor equation is an example of a normal equation that can be placed in an equation section. Declaration equations are usually given as part of declarations of fixed or parameter constants, for example:

```
constant Integer one = 1;
parameter Real mass = 22.5;
```

An equation always holds, which means that the mass in the above example never changes value during simulation. It is also possible to specify a declaration equation for a normal variable, e.g.:

```
Real speed = 72.4;
```

However, this does not make much sense since it will constrain the variable to have the same value throughout the computation, effectively behaving as a constant. Therefore a declaration equation is quite different from a variable initializer in other languages.

Concerning attribute assignments, these are typically specified using modification equations. For example, if we need to specify an initial value for a variable, meaning its value at the start of the computation, then we give an attribute equation for the start attribute of the variable, e.g.:

```
Real speed(start=72.4);
```

See also Chapter 8, page 237, for a complete overview of equations in Modelica.

2.6.1 Repetitive Equation Structures

Before reading this section you might want to take a look at Section 2.13 about arrays, page 44, and Section 2.14.2 about statements and algorithmic for-loops, page 46.

Sometimes there is a need to conveniently express sets of equations that have a regular, i.e., repetitive structure. Often this can be expressed as array equations, including references to array elements denoted using square bracket notation[9]. However, for the more general case of repetitive equation structures Modelica provides a loop construct. Note that this is not a loop in the algorithmic sense of the word—it is rather a shorthand notation for expressing a set of equations.

For example, consider an equation for a polynomial expression:

```
y = a[1]+a[2]*x + a[3]*x^2 + ... + a[n+1]*x^n
```

The polynomial equation can be expressed as a set of equations with regular structure in Modelica, with `y` equal to the scalar product of the vectors `a` and `xpowers`, both of length `n+1`:

```
xpowers[1]  = 1;
xpowers[2]  = xpowers[1]*x;
xpowers[3]  = xpowers[2]*x;
...
xpowers[n+1]  = xpowers[n]*x;
y = a * xpowers;
```

The regular set of equations involving `xpowers` can be expressed more conveniently using the `for` loop notation:

```
for i in 1:n loop
  xpowers[i+1] = xpowers[i]*x;
end for;
```

In this particular case a vector equation provides an even more compact notation:

```
xpowers[2:n+1] = xpowers[1:n]*x;
```

Here the vectors `x` and `xpowers` have length `n+1`. The colon notation `2:n+1` means extracting a vector of length `n`, starting from element 2 up to and including element `n+1`.

2.6.2 Partial Differential Equations

Partial differential equations (abbreviated PDEs) contain derivatives with respect to other variables than time, for example of spatial Cartesian coordinates such as *x* and *y*. Models of phenomena such as heat flow or fluid flow typically involve PDEs. At the time of this writing PDE functionality is not part of the official Modelica language, but is in the process of being included. See Section 8.5, page 258, for an overview of the most important current design proposals which to some extent have been evaluated in test implementations.

2.7 Acausal Physical Modeling

Acausal modeling is a declarative modeling style, meaning modeling based on equations instead of assignment statements. Equations do not specify which variables are inputs and which are outputs, whereas in assignment statements variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equation-based models is unspecified and becomes fixed only when the corresponding equation systems are solved. This is called *acausal modeling*. The term *physical modeling* reflects the fact that *acausal modeling* is very well suited for representing the *physical structure* of modeled systems.

The main advantage with acausal modeling is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The data flow context is defined by stating which variables are needed as *outputs*, and which are external *inputs* to the simulated system.

---

[9] For more information regarding arrays see Chapter 7, page 207.

The acausality of Modelica library classes makes these more reusable than traditional classes containing assignment statements where the `input`-`output` causality is fixed.

2.7.1 Physical Modeling vs. Block-Oriented Modeling

To illustrate the idea of acausal physical modeling we give an example of a simple electrical circuit (Figure 2-7). The connection diagram[10] of the electrical circuit shows how the components are connected. It may be drawn with component placements to roughly correspond to the physical layout of the electrical circuit on a printed circuit board. The physical connections in the real circuit correspond to the logical connections in the diagram. Therefore the *term physical modeling* is quite appropriate.

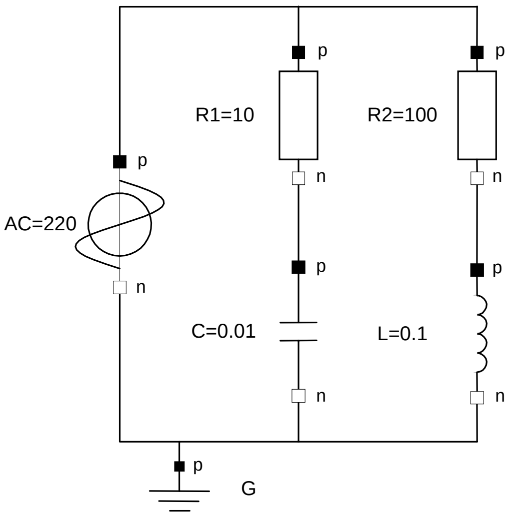


Figure 2-7. Connection diagram of the acausal simple circuit model.

The Modelica `SimpleCircuit` model below directly corresponds to the circuit depicted in the connection diagram of Figure 2-7. Each graphic object in the diagram corresponds to a declared instance in the simple circuit model. The model is acausal since no signal flow, i.e., cause-and-effect flow, is specified. Connections between objects are specified using the `connect` equation construct, which is a special syntactic form of equation that we will examine later. The classes `Resistor`, `Capacitor`, `Inductor`, `VsourceAC`, and `Ground` will be presented in more detail on pages 40 to 43.

```
model SimpleCircuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  Ground    G;
equation
  connect(AC.p, R1.p);    // Capacitor circuit
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);    // Inductor circuit
  connect(R2.n, L.p);
  connect(L.n,  C.n);
  connect(AC.n, G.p);     // Ground
```

[10] A connection diagram emphasizes the connections between components of a model, whereas a composition diagram specifies which components a model is composed of, their subcomponents, etc. A class diagram usually depicts inheritance and composition relations.

```
end SimpleCircuit;
```

As a comparison we show the same circuit modeled using causal block-oriented modeling depicted as a diagram in Figure 2-8. Here the physical topology is lost—the structure of the diagram has no simple correspondence to the structure of the physical circuit board. This model is causal since the signal flow has been deduced and is clearly shown in the diagram. Even for this simple example the analysis to convert the intuitive physical model to a causal block-oriented model is nontrivial. Another disadvantage is that the resistor representations are context dependent. For example, the resistors `R1` and `R2` have different definitions, which makes reuse of model library components hard. Furthermore, such system models are usually hard to maintain since even small changes in the physical structure may result in large changes to the corresponding block-oriented system model.

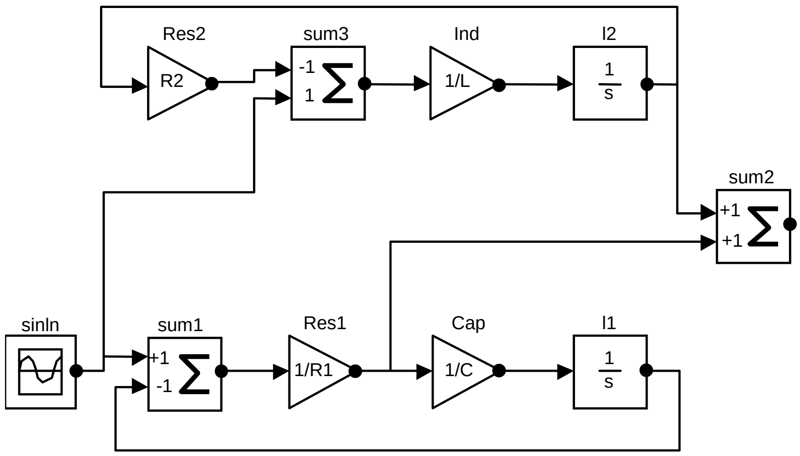


Figure 2-8 The simple circuit model using causal block-oriented modeling with explicit signal flow.

2.8 The Modelica Software Component Model

For a long time, software developers have looked with envy on hardware system builders, regarding the apparent ease with which reusable hardware components are used to construct complicated systems. With software there seems too often to be a need or tendency to develop from scratch instead of reusing components. Early attempts at software components include procedure libraries, which unfortunately have too limited applicability and low flexibility. The advent of object-oriented programming has stimulated the development of software component frameworks such as CORBA, the Microsoft COM/DCOM component object model, and JavaBeans. These component models have considerable success in certain application areas, but there is still a long way to go to reach the level of reuse and component standardization found in hardware industry.

The reader might wonder what all this has to do with Modelica. In fact, Modelica offers quite a powerful software component model that is on par with hardware component systems in flexibility and potential for reuse. The key to this increased flexibility is the fact that Modelica classes are based on equations. What is a software component model? It should include the following three items:

1. Components
2. A connection mechanism
3. A component framework

Components are connected via the connection mechanism, which can be visualized in connection diagrams. The component framework realizes components and connections, and ensures that communication works and constraints are maintained over the connections. For systems composed of acausal components the direction of data flow, i.e., the causality is automatically deduced by the compiler at composition time.

See also Chapter 5, page 145, for a complete overview of components, connectors, and connections.

2.8.1 Components

Components are simply instances of Modelica classes. Those classes should have well-defined interfaces, sometimes called ports, in Modelica called connectors, for communication and coupling between a component and the outside world.

A component is modeled independently of the environment where it is used, which is essential for its reusability. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, should be allowed. However, in Modelica access of component data via dot notation is also possible. A component may internally consist of other connected components, i.e., hierarchical modeling.

2.8.2 Connection Diagrams

Complex systems usually consist of large numbers of connected components, of which many components can be hierarchically decomposed into other components through several levels. To grasp this complexity, a pictorial representation of components and connections is quite important. Such graphic representation is available as connection diagrams, of which a schematic example is shown in Figure 2-9. We have earlier presented a connection diagram of a simple circuit in Figure 2-7.

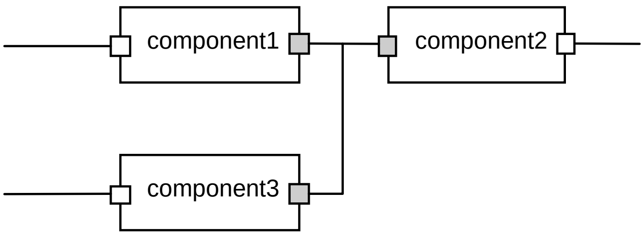


Figure 2-9. Schematic picture of a connection diagram for components.

Each rectangle in the diagram example represents a physical component, e.g., a resistor, a capacitor, a transistor, a mechanical gear, a valve, etc. The connections represented by lines in the diagram correspond to real, physical connections. For example, connections can be realized by electrical wires, by the mechanical connections, by pipes for fluids, by heat exchange between components, etc. The connectors, i.e., interface points, are shown as small square dots on the rectangle in the diagram. Variables at such interface points define the interaction between the component represented by the rectangle and other components.

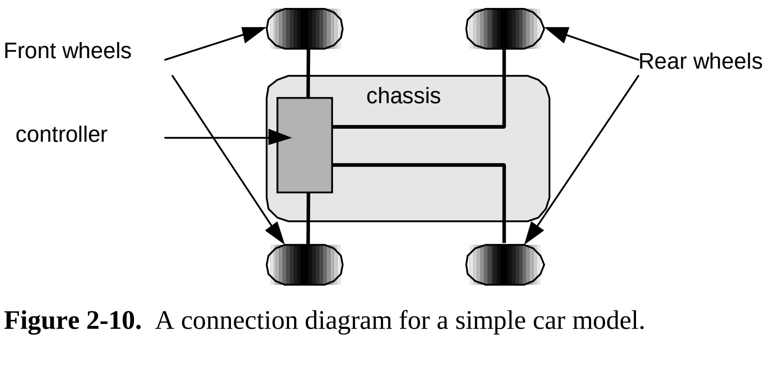


Figure 2-10. A connection diagram for a simple car model.

A simple car example of a connection diagram for an application in the mechanical domain is shown in Figure 2-10.

The simple car model below includes variables for subcomponents such as wheels, chassis, and control unit. A "comment" string after the class name briefly describes the class. The wheels are connected to both the chassis and the controller. Connect equations are present, but are not shown in this partial example.

```
class Car  "A car class to combine car components"
  Wheel          w1,w2,w3,w4  "Wheel one to four";
  Chassis        chassis      "Chassis";
  CarController  controller   "Car controller";
  ...
end Car;
```

2.8.3 Connectors and Connector Classes

Modelica connectors are instances of connector classes, which define the variables that are part of the communication interface that is specified by a connector. Thus, connectors specify external interfaces for interaction.

For example, `Pin` is a connector class that can be used to specify the external interfaces for electrical components (Figure 2-11) that have pins. The types `Voltage` and `Current` used within `Pin` are the same as `Real`, but with different associated units. From the Modelica language point of view the types `Voltage` and `Current` are similar to `Real`, and are regarded as having equivalent types. Checking unit compatibility within equations is optional.

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

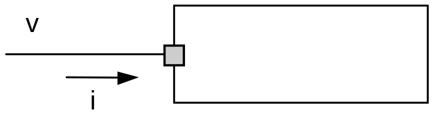


Figure 2-11. A component with one electrical `Pin` connector.

The `Pin` connector class below contains two variables. The `flow` prefix on the second variable indicates that this variable represents a flow quantity, which has special significance for connections as explained in the next section.

```
connector Pin
  Voltage      v;
  flow Current  i;
end Pin;
```

2.8.4 Connections

Connections between components can be established between connectors of equivalent type. Modelica supports equation-based acausal connections, which means that connections are realized as equations. For acausal connections, the direction of data flow in the connection need not be known. Additionally, causal connections can be established by connecting a connector with an `output` attribute to a connector declared as `input`.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are nonflow (default), or declared using the `flow` prefix:

1. Equality coupling, for nonflow variables, according to Kirchhoff's first law.
2. Sum-to-zero coupling, for flow variables, according to Kirchhoff's current law.

For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

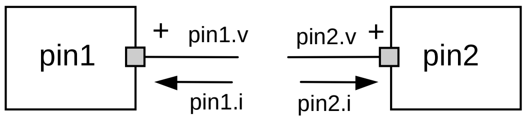


Figure 2-12. Connecting two components that have electrical pins.

Connection equations are used to connect instances of connection classes. A connection equation `connect(pin1,pin2)`, with `pin1` and `pin2` of connector class `Pin`, connects the two pins (Figure 2-12) so that they form one node. This produces two equations, namely:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's second law, saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

See also Section 5.3, page 148, for a complete description of this kind of explicit connections. We should also mention the concept of *implicit connections*, e.g. useful to model force fields, which is represented by the Modelica `inner`/`outer` construct and described in more detail in Section 5.8, page 173.

2.9 Partial Classes

A common property of many electrical components is that they have two pins. This means that it is useful to define a "blueprint" model class, e.g., called `TwoPin`, that captures this common property. This is a *partial class* since it does not contain enough equations to completely specify its physical behavior, and is therefore prefixed by the keyword `partial`. Partial classes are usually known as *abstract classes* in other object-oriented languages.

```
partial class TwoPin[11] "Superclass of elements with two electrical pins"
  Pin      p, n;
  Voltage  v;
  Current  i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

The `TwoPin` class has two pins, `p` and `n`, a quantity `v` that defines the voltage drop across the component, and a quantity `i` that defines the current into pin `p`, through the component, and out from pin `n`

[11] This `TwoPin` class is referred to by the name `Modelica.Electrical.Analog.Interfaces.OnePort` in the Modelica standard library since this is the name used by electrical modeling experts. Here we use the more intuitive name `TwoPin` since the class is used for components with two physical ports and not one. The `OnePort` naming is more understandable if it is viewed as denoting composite ports containing two subports.

(Figure 2-13). It is useful to label the pins differently, e.g., `p` and `n`, and using graphics, e.g. filled and unfilled squares respectively, to obtain a well-defined sign for `v` and `i` although there is no physical difference between these pins in reality.

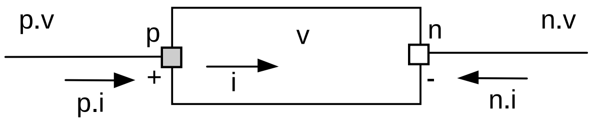


Figure 2-13. Generic `TwoPin` class that describes the general structure of simple electrical components with two pins.

The equations define generic relations between quantities of simple electrical components. In order to be useful, a constitutive equation must be added that describes the specific physical characteristics of the component.

2.9.1 Reuse of Partial Classes

Given the generic partial class `TwoPin`, it is now straightforward to create the more specialized `Resistor` class by adding a constitutive equation:

```
R*i = v;
```

This equation describes the specific physical characteristics of the relation between voltage and current for a resistor (Figure 2-14).

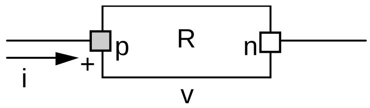


Figure 2-14. A resistor component.

```
class Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R(unit="Ohm") "Resistance";
equation
  R*i = v;
end Resistor;
```

A class for electrical capacitors can also reuse `TwoPin` in a similar way, adding the constitutive equation for a capacitor (Figure 2-15).

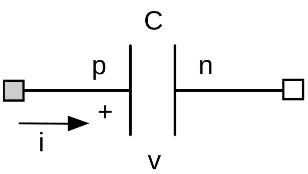


Figure 2-15. A capacitor component.

```
class Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  parameter Real C(Unit="F") "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```

During system simulation the variables `i` and `v` specified in the above components evolve as functions of time. The solver of differential equations computes the values of $v(t)$ and $i(t)$ (where t is `time`) such that $C \cdot \dot{v}(t) = i(t)$ for all values of t, fulfilling the constitutive equation for the capacitor.

2.10 Component Library Design and Use

In a similar way as we previously created the resistor and capacitor components, additional electrical component classes can be created, forming a simple electrical component library that can be used for application models such as the `SimpleCircuit` model. Component libraries of reusable components are actually the key to effective modeling of complex systems.

2.11 Example: Electrical Component Library

Below we show an example of designing a small library of electrical components needed for the simple circuit example, as well as the equations that can be extracted from these components.

2.11.1 Resistor

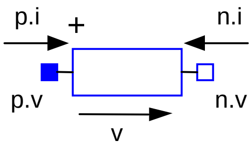


Figure 2-16. Resistor component.

Four equations can be extracted from the resistor model depicted in Figure 2-14 and Figure 2-16. The first three originate from the inherited `TwoPin` class, whereas the last is the constitutive equation of the resistor.

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
v = R*i
```

2.11.2 Capacitor

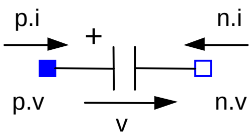


Figure 2-17. Capacitor component.

The following four equations originate from the capacitor model depicted in Figure 2-15 and Figure 2-17, where the last equation is the constitutive equation for the capacitor.

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
i = C * der(v)
```

2.11.3 Inductor

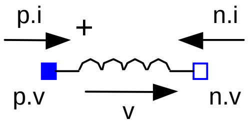


Figure 2-18. Inductor component.

The inductor class depicted in Figure 2-18 and shown below gives a model for ideal electrical inductors.

```
class Inductor "Ideal electrical inductor"
  extends TwoPin;
  parameter Real L(unit="H") "Inductance";
equation
  v = L*der(i);
end Inductor;
```

These equations can be extracted from the inductor class, where the first three come from `TwoPin` as usual and the last is the constitutive equation for the inductor.

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
v = L * der(i)
```

2.11.4 Voltage Source

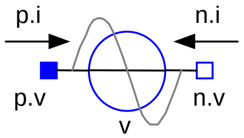


Figure 2-19. Voltage source component `VsourceAC`, where `v(t) = VA*sin(2*PI*f*time)`.

A class `VsourceAC` for the sin-wave voltage source to be used in our circuit example is depicted in Figure 2-19 and can be defined as below. This model as well as other Modelica models specify behavior that evolves as a function of time. Note that a predefined variable `time` is used. In order to keep the example simple the constant `PI` is explicitly declared even though it is usually imported from the Modelica standard library.

```
class VsourceAC "Sin-wave voltage source"
  extends TwoPin;
  parameter Voltage  VA            = 220 "Amplitude";
  parameter Real     f(unit="Hz") = 50  "Frequency";
  constant  Real     PI            = 3.141592653589793;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```

In this `TwoPin`-based model, four equations can be extracted from the model, of which the first three are inherited from `TwoPin`:

```
0 = p.i + n.i
v = p.v - n.v
i = p.i
v = VA*sin(2*PI*f*time)
```

2.11.5 Ground

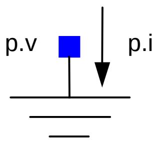


Figure 2-20. Ground component.

Finally, we define a class for ground points that can be instantiated as a reference value for the voltage levels in electrical circuits. This class has only one pin (Figure 2-20).

```
class Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;
```

A single equation can be extracted from the `Ground` class.

```
p.v = 0
```

2.12 The Simple Circuit Model

Having collected a small library of simple electrical components we can now put together the simple electrical circuit shown previously and in Figure 2-21.

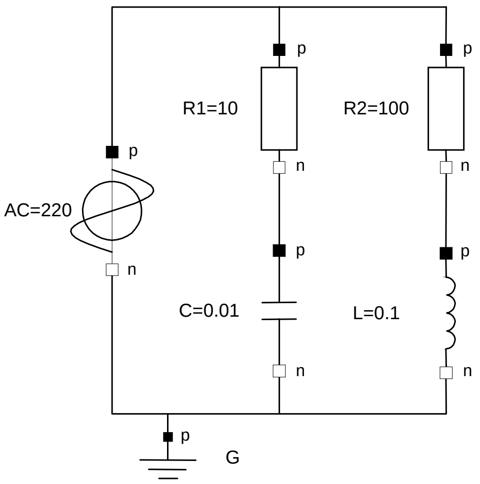


Figure 2-21. The simple circuit model.

The two resistor instances `R1` and `R2` are declared with modification equations for their respective resistance parameter values. Similarly, an instance `C` of the capacitor and an instance `L` of the inductor are declared with modifiers for capacitance and inductance respectively. The voltage source `AC` and the ground instance `G` have no modifiers. Connect equations are provided to connect the components in the circuit.

```
class SimpleCircuit
  Resistor   R1(R=10);
  Capacitor  C(C=0.01);
  Resistor   R2(R=100);
  Inductor   L(L=0.1);
  VsourceAC  AC;
  Ground     G;
```

```
equation
  connect(AC.p, R1.p);    // 1, Capacitor circuit
  connect(R1.n, C.p);     //     Wire 2
  connect(C.n, AC.n);     //     Wire 3
  connect(R1.p, R2.p);    // 2, Inductor circuit
  connect(R2.n, L.p);     //     Wire 5
  connect(L.n,  C.n);     //     Wire 6
  connect(AC.n, G.p);     // 7, Ground
end SimpleCircuit;
```

2.13 Arrays

An array is a collection of variables all of the same type. Elements of an array are accessed through simple integer indexes, ranging from a lower bound of 1 to an upper bound being the size of the respective dimension. An array variable can be declared by appending dimensions within square brackets after a class name, as in Java, or after a variable name, as in the C language. For example:

```
Real[3]      positionvector = {1,2,3};
Real[3,3]    identitymatrix = {{1,0,0}, {0,1,0}, {0,0,1}};
Real[3,3,3] arr3d;
```

This declares a three-dimensional position vector, a transformation matrix, and a three-dimensional array. Using the alternative syntax of attaching dimensions after the variable name, the same declarations can be expressed as:

```
Real  positionvector[3] = {1,2,3};
Real  identitymatrix[3,3] = {{1,0,0}, {0,1,0}, {0,0,1}};
Real  arr3d[3,3,3];
```

In the first two array declarations, declaration equations have been given, where the array constructor {} is used to construct array values for defining positionvector and identitymatrix. Indexing of an array `A` is written `A[i,j,...]`, where 1 is the lower bound and `size(A,k)` is the upper bound of the index for the *k*th dimension. Submatrices can be formed by utilizing the `:` notation for index ranges, for example, `A[i1:i2, j1:j2]`, where a range `i1:i2` means all indexed elements starting with `i1` up to and including `i2`.

Array expressions can be formed using the arithmetic operators +, -, *, and /, since these can operate on either scalars, vectors, matrices, or (when applicable) multidimensional arrays with elements of type `Real` or `Integer`. The multiplication operator * denotes scalar product when used between vectors, matrix multiplication when used between matrices or between a matrix and a vector, and element-wise multiplication when used between an array and a scalar. As an example, multiplying `positionvector` by the scalar `2` is expressed by:

```
positionvector*2
```

which gives the result:

```
{2,4,6}
```

In contrast to Java, arrays of dimensionality > 1 in Modelica are always rectangular as in Matlab or Fortran.

A number of built-in array functions are available, of which a few are shown in the table below.

`transpose(A)`	Permutes the first two dimensions of array A.
`zeros(n1,n2,n3,...)`	Returns an n_1 x n_2 x n_3 x... zero-filled integer array.
`ones(n1,n2,n3,...)`	Returns an n_1 x n_2 x n_3 x ... one-filled integer array.
`fill(s,n1,n2,n3, ...)`	Returns the n_1 x n_2 x n_3 x ... array with all elements filled with the

	value of the scalar expression s.
`min(A)`	Returns the smallest element of array expression A.
`max(A)`	Returns the largest element of array expression A.
`sum(A)`	Returns the sum of all the elements of array expression A.

A scalar Modelica function of a scalar argument is automatically generalized to be applicable also to arrays element-wise. For example, if A is a vector of real numbers, then `cos(A)` is a vector where each element is the result of applying the function `cos` to the corresponding element in A. For example:

```
cos({1, 2, 3})  =  {cos(1), cos(2), cos(3)}
```

General array concatenation can be done through the array concatenation operator `cat(k,A,B,C,...)` that concatenates the arrays `A,B,C,...` along the `k`:th dimension. For example, `cat(1,{2,3}, {5,8,4})` gives the result `{2,3,5,8,4}`.

The common special cases of concatenation along the first and second dimensions are supported through the special syntax forms [`A;B;C;...`] and [`A,B,C,...`] respectively. Both of these forms can be mixed. In order to achieve compatibility with Matlab array syntax, being a de facto standard, scalar and vector arguments to these special operators are promoted to become matrices before performing the concatenation. This gives the effect that a matrix can be constructed from scalar expressions by separating rows by semicolons and columns by commas. The example below creates an $m \times n$ matrix:

```
[expr11, expr12, ... expr1n;
expr21, expr22, ... expr2n;
…
exprm1, exprm2, ... exprmn]
```

It is instructive to follow the process of creating a matrix from scalar expressions using these operators. For example:

```
[1,2;
 3,4]
```

First each scalar argument is promoted to become a matrix, giving:

```
[{{1}}, {{2}};
 {{3}}, {{4}}]
```

Since [... , ...] for concatenation along the second dimension has higher priority than [... ; ...], which concatenates along the first dimension, the first concatenation step gives:

```
[{{1, 2}};
 {{3, 4}}]
```

Finally, the row matrices are concatenated giving the desired 2×2 matrix:

```
 {{1, 2},
  {3, 4}}
```

The special case of just one scalar argument can be used to create a 1×1 matrix. For example:

```
[1]
```

gives the matrix:

```
{{1}}
```

See also Chapter 7, page 207, for a complete overview of Modelica arrays.

2.14 Algorithmic Constructs

Even though equations are eminently suitable for modeling physical systems and for a number of other tasks, there are situations where nondeclarative algorithmic constructs are needed. This is typically the case for algorithms, i.e., procedural descriptions of how to carry out specific computations, usually consisting of a number of statements that should be executed in the specified order.

2.14.1 Algorithms

In Modelica, algorithmic statements can occur only within algorithm sections, starting with the keyword `algorithm`. Algorithm sections may also be called algorithm equations, since an algorithm section can be viewed as a group of equations involving one or more variables, and can appear among equation sections. Algorithm sections are terminated by the appearance of one of the keywords `equation`, `public`, `protected`, `algorithm`, or `end`.

```
algorithm
  ...
  <statements>
  ...
  <some other keyword>
```

An algorithm section embedded among equation sections can appear as below, where the example algorithm section contains three assignment statements.

```
equation
  x = y*2;
  z = w;
algorithm
  x1 := z+x;
  x2 := y-5;
  x1 := x2+y;
equation
  u = x1+x2;
  ...
```

Note that the code in the algorithm section, sometimes denoted algorithm equation, uses the values of certain variables from outside the algorithm. These variables are so called *input variables* to the algorithm—in this example `x`, `y`, and `z`. Analogously, variables assigned values by the algorithm define the *outputs of the algorithm*—in this example `x1` and `x2`. This makes the semantics of an algorithm section quite similar to a function with the algorithm section as its body, and with `input` and `output` formal parameters corresponding to inputs and outputs as described above.

See also Chapter 9, page 283, regarding algorithms and functions.

2.14.2 Statements

In addition to assignment statements, which were used in the previous example, three other kinds of "algorithmic" statements are available in Modelica: `if-then-else` statements, for-loops, and while-loops. The summation below uses both a while-loop and an if-statement, where `size (a,1)` returns the size of the first dimension of array `a`. The elseif- and else-parts of if-statements are optional.

```
sum := 0;
n := size(a,1);
while n>0 loop
  if a[n]>0 then
    sum := sum + a[n];
```

```
    elseif a[n] > -1 then
      sum := sum - a[n] -1;
    else
      sum := sum - a[n];
    end if;
    n := n-1;
  end while;
```

Both for-loops and while-loops can be immediately terminated by executing a break-statement inside the loop. Such a statement just consists of the keyword `break` followed by a semicolon.

Consider once more the computation of the polynomial presented in Section 2.6.1 on repetitive equation structures, page 33.

```
y := a[1]+a[2]*x + a[3]*x^1 + ... + a[n+1]*x^n;
```

When using equations to model the computation of the polynomial it was necessary to introduce an auxliliary vector `xpowers` for storing the different powers of `x`. Alternatively, the same computation can be expressed as an algorithm including a for-loop as below. This can be done without the need for an extra vector—it is enough to use a scalar variable `xpower` for the most recently computed power of `x`.

```
algorithm
  y := 0;
  xpower := 1;
  for i in 1:n+1 loop
    y := y + a[i]*xpower;
    xpower := xpower*x;
  end for;
   ...
```

See Section 9.2.3, page 287, for descriptions of statement constructs in Modelica.

2.14.3 Functions

Functions are a natural part of any mathematical model. A number of mathematical functions like `abs`, `sqrt`, `mod`, etc. are predefined in the Modelica language whereas others such as `sin`, `cos`, `exp`, etc. are available in the Modelica standard math library `Modelica.Math`. The arithmetic operators +, -, *, / can be regarded as functions that are used through a convenient operator syntax. Thus it is natural to have user-defined mathematical functions in the Modelica language. The body of a Modelica function is an algorithm section that contains procedural algorithmic code to be executed when the function is called. Formal parameters are specified using the `input` keyword, whereas results are denoted using the `output` keyword. This makes the syntax of function definitions quite close to Modelica block class definitions.

Modelica functions are *mathematical functions*, i.e., without global side-effects and with no memory. A Modelica function always returns the same results given the same arguments. Below we show the algorithmic code for polynomial evaluation in a function named `polynomialEvaluator`.

```
function polynomialEvaluator
  input  Real a[:];       // Array, size defined at function call time
  input  Real x := 1.0;   // Default value 1.0 for x
  output Real y;
protected
  Real   xpower;
algorithm
  y := 0;
  xpower := 1;
  for i in 1:size(a,1) loop
    y := y + a[i]*xpower;
```

```
    xpower := xpower*x;
  end for;
end polynomialEvaluator;
```

Functions are usually called with positional association of actual arguments to formal parameters. For example, in the call below the actual argument `{1,2,3,4}` becomes the value of the coefficient vector `a`, and `21` becomes the value of the formal parameter `x`. Modelica function parameters are read-only, i.e., they may not be assigned values within the code of the function. When a function is called using positional argument association, the number of actual arguments and formal parameters must be the same. The types of the actual argument expressions must be compatible with the declared types of the corresponding formal parameters. This allows passing array arguments of arbitrary length to functions with array formal parameters with unspecified length, as in the case of the input formal parameter `a` in the `polynomialEvaluator` function.

```
p = polynomialEvaluator({1, 2, 3, 4}, 21);
```

The same call to the function `polynomialEvaluator` can instead be made using named association of actual arguments to formal parameters, as in the next example. This has the advantage that the code becomes more self-documenting as well as more flexible with respect to code updates.

For example, if all calls to the function `polynomialEvaluator` are made using named parameter association, the order between the formal parameters `a` and `x` can be changed, and new formal parameters with default values can be introduced in the function definitions without causing any compilation errors at the call sites. Formal parameters with default values need not be specified as actual arguments unless those parameters should be assigned values different from the defaults.

```
p = polynomialEvaluator(a={1, 2, 3, 4}, x=21);
```

Functions can have multiple results. For example, the function `f` below has three result parameters declared as three formal output parameters `r1`, `r2`, and `r3`.

```
function f
  input Real x;
  input Real y;
  output Real r1;
  output Real r2;
  output Real r3;
  ...
end f;
```

Within algorithmic code multiresult functions may be called only in special assignment statements, as the one below, where the variables on the left-hand side are assigned the corresponding function results.

```
(a, b, c) := f(1.0, 2.0);
```

In equations a similar syntax is used:

```
(a, b, c) = f(1.0, 2.0);
```

A function is returned from by reaching the end of the function or by executing a return-statement inside the function body.

See also Section 9.3, page 298, for more information regarding functions.

2.14.4 Function and Operator Overloading

Function and operator overloading allows several definitions of the same function or operator, but with a different set of input formal parameter types for each definition. This allows, e.g., to define operators such as addition, multiplication, etc., of complex numbers, using the ordinary + and * operators but with new definitions, or provide several definitions of a `solve` function for linear matrix equation solution for different matrix representations such as standard dense matrices, sparse matrices, symmetric matrices, etc. Such functionality is not yet part of the official Modelica language at the time of this writing, but is on its way into the language, and test implementations are available. See Section 9.5, page 322 for more information regarding this topic.

2.14.5 External Functions

It is possible to call functions defined outside of the Modelica language, implemented in C or Fortran. If no external language is specified the implementation language is assumed to be C. The body of an external function is marked with the keyword `external` in the Modelica external function declaration.

```
function log
  input Real x;
  output Real y;
external
end log;
```

The external function interface supports a number of advanced features such as in—out parameters, local work arrays, external function argument order, explicit specification of row-major versus column-major array memory layout, etc. For example, the formal parameter `Ares` corresponds to an in—out parameter in the external function `leastSquares` below, which has the value `A` as input default and a different value as the result. It is possible to control the ordering and usage of parameters to the function external to Modelica. This is used below to explicitly pass sizes of array dimensions to the Fortran routine called `dgels`. Some old-style Fortran routines like `dgels` need work arrays, which is conveniently handled by local variable declarations after the keyword `protected`.

```
function leastSquares "Solves a linear least squares problem"
  input  Real A[:,:];
  input  Real B[:,:];
  output Real Ares[size(A,1),size(A,2)] := A;
    //Factorization is returned in Ares for later use
  output Real x[size(A,2),size(B,2)];
protected
  Integer lwork = min(size(A,1),size(A,2))+
                  max(max(size(A,1),size(A,2)),size(B,2))*32;
  Real work[lwork];
  Integer info;
  String transposed="NNNN"; // Workaround for passing CHARACTER data to
                            // Fortran routine
  external "FORTRAN 77"
  dgels(transposed, 100, size(A,1), size(A,2), size(B,2), Ares,
        size(A,1), B, size(B,1), work, lwork, info);
end leastSquares;
```

See also Section 9.4, page 311, regarding external functions.

2.14.6 Algorithms Viewed as Functions

The function concept is a basic building block when defining the semantics or meaning of programming language constructs. Some programming languages are completely defined in terms of mathematical functions. This makes it useful to try to understand and define the semantics of algorithm sections in Modelica in terms of functions. For example, consider the algorithm section below, which occurs in an equation context:

```
algorithm
  y := x;
```

```
  z := 2*y;
  y := z+y;
  ...
```

This algorithm can be transformed into an equation and a function as below, without changing its meaning. The equation equates the output variables of the previous algorithm section with the results of the function `f`. The function `f` has the inputs to the algorithm section as its input formal parameters and the outputs as its result parameters. The algorithmic code of the algorithm section has become the body of the function `f`.

```
(y,z) = f(x);
...
function f
  input  Real x;
  output Real y,z;
algorithm
  y := x;
  z := 2*y;
  y := z+y;
end f;
```

2.15 Discrete Event and Hybrid Modeling

Macroscopic physical systems in general evolve continuously as a function of time, obeying the laws of physics. This includes the movements of parts in mechanical systems, current and voltage levels in electrical systems, chemical reactions, etc. Such systems are said to have continuous dynamics.

On the other hand, it is sometimes beneficial to make the approximation that certain system components display discrete behavior, i.e., changes of values of system variables may occur instantaneously and discontinuously at specific points in time.

In the real physical system the change can be very fast, but not instantaneous. Examples are collisions in mechanical systems, e.g., a bouncing ball that almost instantaneously changes direction, switches in electrical circuits with quickly changing voltage levels, valves and pumps in chemical plants, etc. We talk about system components with discrete-time dynamics. The reason to make the discrete approximation is to simplify the mathematical model of the system, making the model more tractable and usually speeding up the simulation of the model several orders of magnitude.

For this reason it is possible to have variables in Modelica models of *discrete-time variability*, i.e., the variables change value only at specific points in time, so-called *events*, and keep their values constant between events, as depicted in Figure 2-22. Examples of discrete-time variables are `Real` variables declared with the prefix `discrete`, or `Integer`, `Boolean`, and `enumeration` variables which are discrete-time by default and cannot be continuous-time.

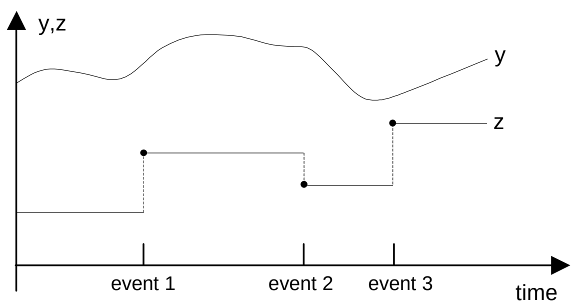


Figure 2-22. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change value both between and at events.

Since the discrete-time approximation can only be applied to certain subsystems, we often arrive at system models consisting of interacting continuous and discrete components. Such a system is called a *hybrid system* and the associated modeling techniques *hybrid modeling*. The introduction of hybrid mathematical models creates new difficulties for their solution, but the disadvantages are far outweighed by the advantages.

Modelica provides two kinds of constructs for expressing hybrid models: conditional expressions or equations to describe discontinuous and conditional models, and when-equations to express equations that are valid only at discontinuities, e.g., when certain conditions become true. For example, if-then-else conditional expressions allow modeling of phenomena with different expressions in different operating regions, as for the equation describing a limiter below.

```
y = if v > limit then limit else v;
```

A more complete example of a conditional model is the model of an ideal diode. The characteristic of a real physical diode is depicted in Figure 2-23, and the ideal diode characteristic in parameterized form is shown in Figure 2-24.

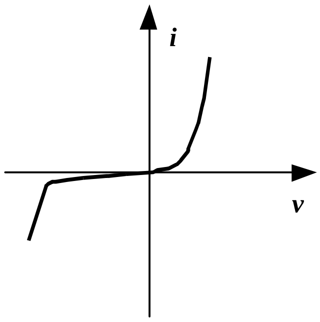


Figure 2-23. Real diode characteristic.

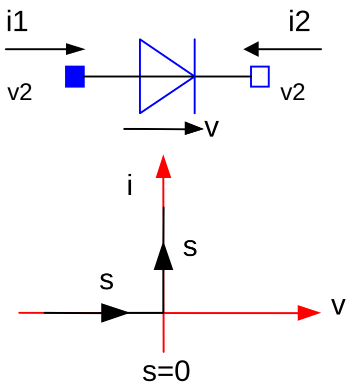


Figure 2-24. Ideal diode characteristic.

Since the voltage level of the ideal diode would go to infinity in an ordinary voltage-current diagram, a parameterized description is more appropriate, where both the voltage `v` and the current `i`, same as `i1`, are functions of the parameter `s`. When the diode is off no current flows and the voltage is negative, whereas when it is on there is no voltage drop over the diode and the current flows.

```
model Diode "Ideal diode"
  extends TwoPin;
  Real s;
  Boolean off;
equation
```

```
    off = s < 0;
    if off
      then v=s;
      else v=0;   // conditional equations
    end if;
    i = if off then 0 else s;   // conditional expression
  end Diode;
```

When-equations have been introduced in Modelica to express instantaneous equations, i.e., equations that are valid only at certain points in time that, for example, occur at discontinuities when specific conditions become true, so-called *events*. The syntax of when-equations for the case of a vector of conditions is shown below. The equations in the when-equation are activated when at least one of the conditions becomes true, and remain activated only for a time instant of zero duration. A single condition is also possible.

```
when {condition1, condition2, …} then
  <equations>
end when;
```

A bouncing ball is a good example of a hybrid system for which the when-equation is appropriate when modeled. The motion of the ball is characterized by the variable `height` above the ground and the vertical `velocity`. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 2-25. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient of 1 and would not lose any energy at a bounce. A more realistic ball, as the one modeled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

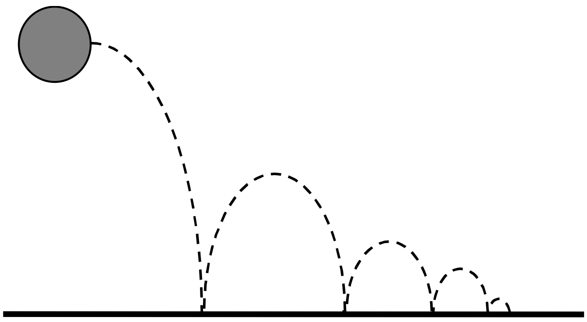


Figure 2-25. A bouncing ball.

The bouncing ball model contains the two basic equations of motion relating height and velocity as well as the acceleration caused by the gravitational force. At the bounce instant the velocity is suddenly reversed and slightly decreased, i.e., `velocity` (after bounce) `= -c*velocity` (before bounce), which is accomplished by the special `reinit` syntactic form of instantaneous equation for reinitialization: `reinit(velocity,-c*pre(velocity))`, which in this case reinitializes the `velocity` variable.

```
model BouncingBall "Simple model of a bouncing ball"
  constant  Real g = 9.81   "Gravity constant";
  parameter Real c = 0.9    "Coefficient of restitution";
  parameter Real radius=0.1 "Radius of the ball";
  Real height(start = 1)    "Height of the ball center";
  Real velocity(start = 0)  "Velocity of the ball";
equation
  der(height) = velocity;
  der(velocity) = -g;
  when height <= radius then
     reinit(velocity,-c*pre(velocity));
  end when;
end BouncingBall;
```

Note that the equations within a when-equation are active only during the instant in time when the condition(s) of the when-equation become `true`, whereas the conditional equations within an if-equation are active as long as the condition of the if-equation is true.

If we simulate this model long enough, the ball will fall through the ground. This strange behavior of the simulation, called shattering, or the Zeno effect (explained in more detail in Section 18.2.5.6, page 685) is due to the limited precision of floating point numbers together with the event detection mechanism of the simulator, and occurs for some (unphysical) models where events may occur infinitely close to each other. The real problem in this case is that the model of the impact is not realistic—the law `new_velocity = -c*velocity` does not hold for very small velocities. A simple fix is to state a condition when the ball falls through the ground and then switch to an equation stating that the ball is lying on the ground. A better but more complicated solution is to switch to a more realistic material model.

See also Section 8.3.4 below on page 245; Section 9.2.9, page 293; and Chapter 13, page 405, regarding discrete and hybrid issues.

2.16 Packages

Name conflicts are a major problem when developing reusable code, for example, libraries of reusable Modelica classes and functions for various application domains. No matter how carefully names are chosen for classes and variables it is likely that someone else is using some name for a different purpose. This problem gets worse if we are using short descriptive names since such names are easy to use and therefore quite popular, making them quite likely to be used in another person's code.

A common solution to avoid name collisions is to attach a short prefix to a set of related names, which are grouped into a package. For example, all names in the X-Windows toolkit have the prefix Xt, and WIN32 is the prefix for the 32-bit Windows API. This works reasonably well for a small number of packages, but the likelihood of name collisions increases as the number of packages grows.

Many programming languages, e.g., Java and Ada as well as Modelica provide a safer and more systematic way of avoiding name collisions through the concept of *package*. A package is simply a container or name space for names of classes, functions, constants, and other allowed definitions. The package name is prefixed to all definitions in the package using standard dot notation. Definitions can be *imported* into the name space of a package.

Modelica has defined the package concept as a restriction and enhancement of the class concept. Thus, inheritance could be used for importing definitions into the name space of another package. However, this gives conceptual modeling problems since inheritance for import is not really a package specialization. Instead, an `import` language construct is provided for Modelica packages. The type name `Voltage` together with all other definitions in `Modelica.SIunits` is imported in the example below, which makes it possible to use it without prefix for declaration of the variable `v`. By contrast, the declaration of the variable `i` uses the fully qualified name `Modelica.SIunits.Ampere` of the type `Ampere`, even though the short version also would have been possible. The fully qualified long name for `Ampere` can be used since it is found using the standard nested lookup of the `Modelica` standard library placed in a conceptual top-level package.

```
package MyPack
  import Modelica.SIunits.*;

  class Foo;
    Voltage  v;
    Modelica.SIunits.Ampere  i;
  end foo;

end MyPack;
```

Importing definitions from one package into another package as in the above example has the drawback that the introduction of new definitions into a package may cause name clashes with definitions in packages using that package. For example, if a definition named `v` is introduced into the package `Modelica.SIunits`, a compilation error would arise in the package `MyPack`.

An alternative solution to the short-name problem that does not have the drawback of possible compilation errors when new definitions are added to libraries, is introducing short convenient name aliases for prefixes instead of long package prefixes. This is possible using the renaming form of `import` statement as in the package `MyPack` below, where the package name `SI` is introduced instead of the much longer `Modelica.SIunits`.

Another disadvantage with the above package is that the `Ampere` type is referred to using standard nested lookup and not via an explicit `import` statement. Thus, in the worst case we may have to do the following in order to find all such dependencies and the declarations they refer to:

- Visually scan the whole source code of the current package, which might be large.
- Search through all packages containing the current package, i.e., higher up in the package hierarchy, since standard nested lookup allows used types and other definitions to be declared anywhere above the current position in the hierarchy.

Instead, a *well-designed package* should state all its dependencies *explicitly* through `import` statements which are easy to find. We can create such a package, e.g., the package `MyPack` below, by adding the prefix `encapsulated` in front of the `package` keyword. This prevents nested lookup outside the package boundary, ensuring that all dependencies on other packages outside the current package have to be explicitly stated as `import` statements. This kind of encapsulated package represents an independent unit of code and corresponds more closely to the package concept found in many other programming languages, e.g., Java or Ada.

```
encapsulated package MyPack
  import SI = Modelica.SIunits;
  import Modelica;

  class Foo;
    SI.Voltage v;
    Modelica.SIunits.Ampere i;
  end Foo;
  ...

end MyPack;
```

See Chapter 10, page 333, for additional details concerning packages and import.

2.17 Annotations

A Modelica annotation is extra information associated with a Modelica model. This additional information is used by Modelica environments, e.g., for supporting documentation or graphical model editing. Most annotations do not influence the execution of a simulation, i.e., the same results should be obtained even if the annotations are removed—but there are exceptions to this rule. The syntax of an annotation is as follows:

annotation(*annotation_elements*)

where *annotation_elements* is a comma-separated list of annotation elements that can be any kind of expression compatible with the Modelica syntax. The following is a resistor class with its associated annotation for the icon representation of the resistor used in the graphical model editor:

```
model Resistor
```

```
    annotation(Icon(coordinateSystem(extent={{-120,-120},{120,120}}),
      graphics = {
        Rectangle(extent=[-70, -30; 70, 30], fillPattern=FillPattern.None),
        Line(points=[-90, 0; -70, 0]),
        ...
    ));
    ...
  end Resistor;
```

Another example is the predefined annotation `choices` used to generate menus for the graphical user interface:

```
annotation(choices(choice=1 "P",  choice=2 "PI",  choice=3 "PID"));
```

The external function annotation `arrayLayout` can be used to explicitly give the layout of arrays, e.g., if it deviates from the defaults `rowMajor` and `columnMajor` order for the external languages C and Fortran 77 respectively.

This is one of the rare cases of an annotation influencing the simulation results, since the wrong array layout annotation obviously will have consequences for matrix computations. An example:

```
annotation(arrayLayout = "columnMajor");
```

See also Chapter 11, page 357.

2.18 Naming Conventions

You may have noticed a certain style of naming classes and variables in the examples in this chapter. In fact, certain naming conventions, described below, are being adhered to. These naming conventions have been adopted in the Modelica standard library, making the code more readable and somewhat reducing the risk for name conflicts. The naming conventions are largely followed in the examples in this book and are recommended for Modelica code in general:

- Type and class names (but usually not functions) always start with an uppercase letter, e.g., `Voltage`.
- Variable names start with a lowercase letter, e.g., `body`, with the exception of some one-letter names such as T for temperature.
- Names consisting of several words have each word capitalized, with the initial word subject to the above rules, e.g., `ElectricCurrent` and `bodyPart`.
- The underscore character is only used at the end of a name, or at the end of a word within a name, to characterize lower or upper indices, e.g., `body_low_up`.
- Preferred names for connector instances in (partial) models are `p` and `n` for positive and negative connectors in electrical components, and name variants containing `a` and `b`, e.g., `flange_a` and `flange_b`, for other kinds of otherwise-identical connectors often occurring in two-sided components.

2.19 Modelica Standard Libraries

Much of the power of modeling with Modelica comes from the ease of reusing model classes. Related classes in particular areas are grouped into packages to make them easier to find.

A special package, called `Modelica`, is a standardized predefined package that together with the Modelica Language is developed and maintained by the Modelica Association. This package is also known as the *Modelica Standard Library*. It provides constants, types, connector classes, partial models, and model classes of components from various application areas, which are grouped into subpackages of the `Modelica` package, known as the Modelica standard libraries.

The following is a subset of the growing set of Modelica standard libraries currently available:

`Modelica.Constants`	Common constants from mathematics, physics, etc.
`Modelica.Icons`	Graphical layout of icon definitions used in several packages.
`Modelica.Math`	Definitions of common mathematical functions.
`Modelica.SIUnits`	Type definitions with SI standard names and units.
`Modelica.Electrical`	Common electrical component models.
`Modelica.Blocks`	Input/output blocks for use in block diagrams.
`Modelica.Mechanics.Translational`	1D mechanical translational components.
`Modelica.Mechanics.Rotational`	1D mechanical rotational components.
`Modelica.Mechanics.MultiBody`	MBS library—3D mechanical multibody models.
`Modelica.Thermal`	Thermal phenomena, heat flow, etc. components.
...	...

Additional libraries are available in application areas such as thermodynamics, hydraulics, power systems, data communication, etc.

The Modelica Standard Library can be used freely for both noncommercial and commercial purposes under the conditions of *The Modelica License* as stated in the front pages of this book. The full documentation as well as the source code of these libraries appear at the Modelica web site.

So far the models presented have been constructed of components from single-application domains. However, one of the main advantages with Modelica is the ease of constructing multidomain models simply by connecting components from different application domain libraries. The DC (direct current) motor depicted in Figure 2-26 is one of the simplest examples illustrating this capability.

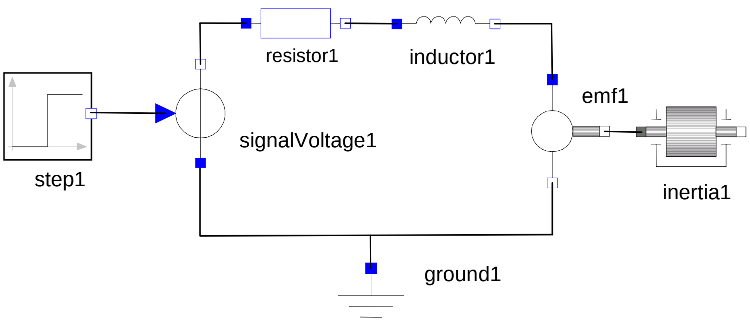


Figure 2-26. A multidomain `DCMotorCircuit` model with mechanical, electrical, and signal block components.

This particular model contains components from the three domains, mechanical, electrical, and signal blocks, corresponding to the libraries `Modelica.Mechanics`, `Modelica.Electrical`, and `Modelica.Blocks`.

Model classes from libraries are particularly easy to use and combine when using a graphical model editor, as depicted in Figure 2-27, where the DC-motor model is being constructed. The left window shows the `Modelica.Mechanics.Rotational` library, from which icons can be dragged and dropped into the central window when performing graphic design of the model.

See also Chapter 16, page 615, for an overview of current Modelica libraries, and Appendix D for some source code.

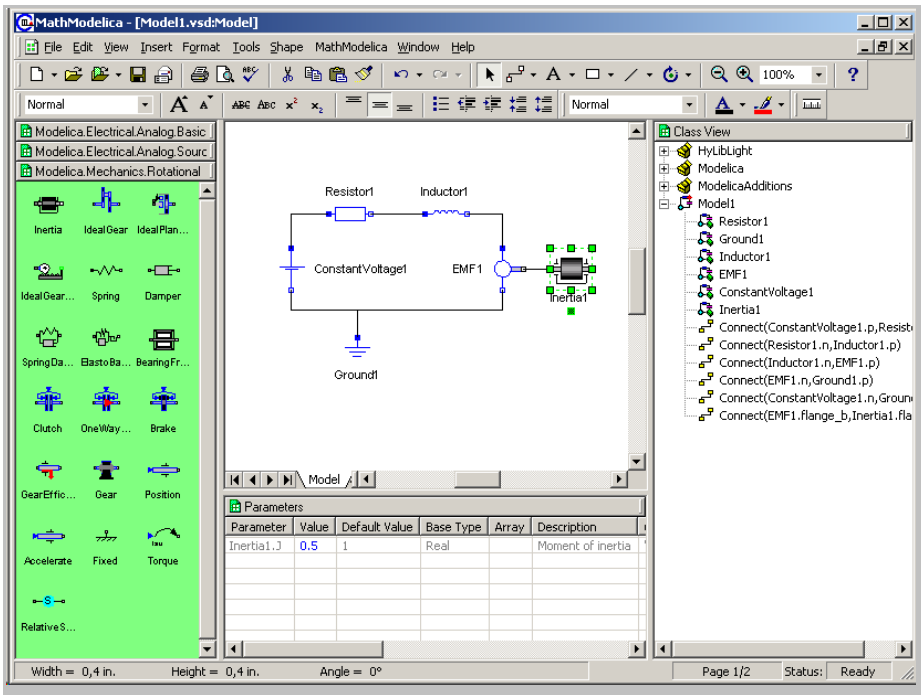


Figure 2-27. Graphical editing of an electrical DC-motor model, with the icons of the `Modelica.Mechanics.Rotational` library in the left window.

2.20 Implementation and Execution of Modelica

In order to gain a better understanding of how Modelica works it is useful to take a look at the process of translation and execution of a Modelica model, which is sketched in Figure 2-28. First the Modelica source code is parsed and converted into an internal representation, usually an abstract syntax tree. This representation is analyzed, type checking is done, classes are inherited and expanded, modifications and instantiations are performed, connect equations are converted to standard equations, etc. The result of this analysis and translation process is a flat set of equations, constants, variables, and function definitions. No trace of the object-oriented structure remains apart from the dot notation within names.

After flattening, all of the equations are topologically sorted according to the data-flow dependencies between the equations. In the case of general differential algebraic equations (DAEs), this is not just sorting, but also manipulation of the equations to convert the coefficient matrix into block lower triangular form, a so-called BLT transformation. Then an optimizer module containing algebraic simplification algorithms, symbolic index reduction methods, etc., eliminates most equations, keeping only a minimal set that eventually will be solved numerically. As a trivial example, if two syntactically equivalent equations appear, only one copy of the equations is kept. Then independent equations in explicit form are converted to assignment statements. This is possible since the equations have been sorted and an execution order has been established for evaluation of the equations in conjunction with the iteration steps of the numeric solver. If a strongly connected set of equations appears, this set is transformed by a symbolic solver, which performs a number of algebraic transformations to simplify the dependencies between the variables. It can sometimes solve a system of differential equations if it has a symbolic solution. Finally, C code is generated, and linked with a numeric equation solver that solves the remaining, drastically reduced, equation system.

The approximations to initial values are taken from the model definition or are interactively specified by the user. If necessary, the user also specifies the parameter values. A numeric solver for differential-algebraic equations (or in simple cases for ordinary differential equations) computes the

Index