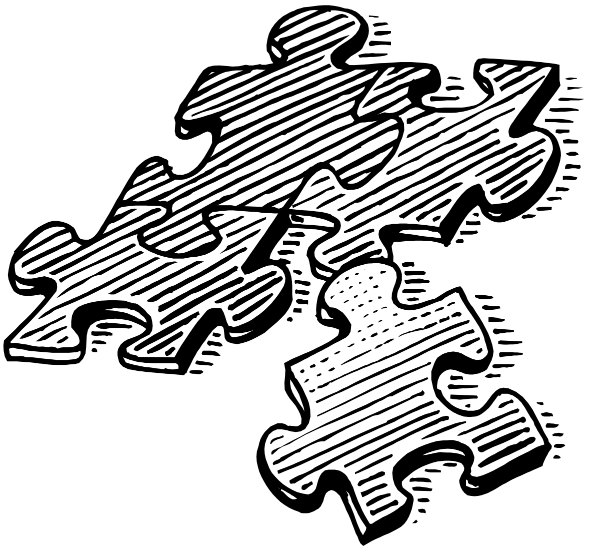
Peter Fritzson

Principles of Object-Oriented Modeling and Simulation with Modelica



Peter Fritzson

Principles of Object-Oriented Modeling and Simulation with Modelica

September 2003

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`.

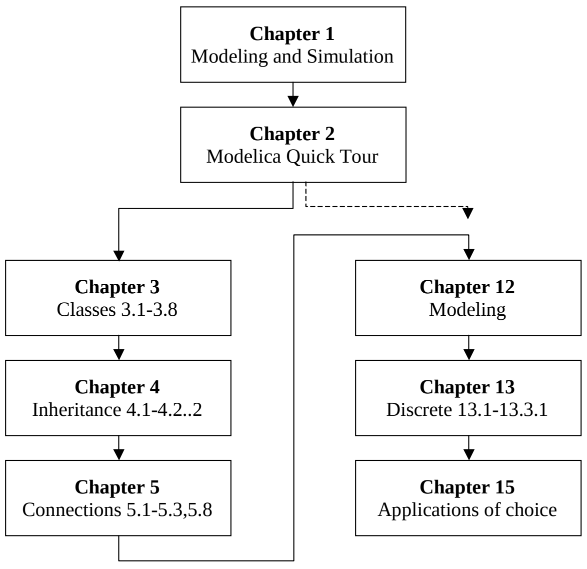
Reading Guide

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

Part I

Introduction

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.

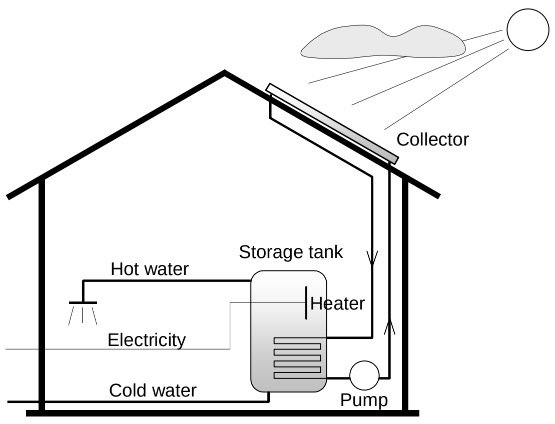


Figure 1-1. A system: a house with solar-heated tap warm water, together with clouds and sunshine.

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.

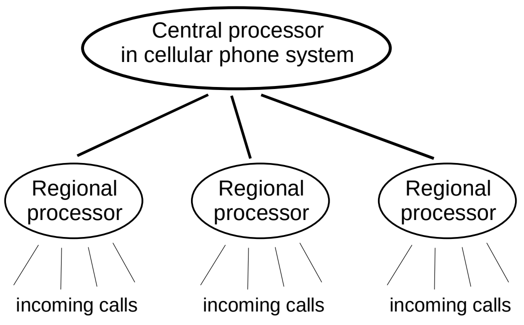


Figure 1-2. A cellular phone system containing a central processor and regional processors to handle incoming calls.

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.

1.3 Simulation

In the previous section we mentioned the possibility of performing "experiments" on models instead of on the real systems corresponding to the models. This is actually one of the main uses of models, and is denoted by the term *simulation*, from the Latin *simulare,* which means to pretend. We define a simulation as follows:

- A *simulation* is an experiment performed on a model.

In analogy with our previous definition of *model*, this definition of simulation does not require the model to be represented in mathematical or computer program form. However, in the rest of this text we will concentrate on *mathematical models*, primarily those which have a computer representable form. The following are a few examples of such experiments or simulations:

- A simulation of an industrial process such as steel or pulp manufacturing, to learn about the behavior under different operating conditions in order to improve the process.
- A simulation of vehicle behavior, e.g. of a car or an airplane, for the purpose of providing realistic operator training.
- A simulation of a simplified model of a packet-switched computer network, to learn about its behavior under different loads in order to improve performance.

It is important to realize that the *experiment description* and *model description* parts of a simulation are conceptually separate entities. On the other hand, these two aspects of a simulation belong together even if they are separate. For example, a model is valid only for a certain class of experiments. It can be useful to define an *experimental frame* associated with the model, which defines the conditions that need to be fulfilled by valid experiments.

If the mathematical model is represented in executable form in a computer, simulations can be performed by *numerical experiments*, or in nonnumerical cases by *computed experiments*. This is a simple and safe way of performing experiments, with the added advantage that essentially all variables of the model are observable and controllable. However, the value of the simulation results is completely dependent on how well the model represents the real system regarding the questions to be answered by the simulation.

Except for experimentation, simulation is the only technique that is generally applicable for analysis of the behavior of arbitrary systems. Analytical techniques are better than simulation, but usually apply only under a set of simplifying assumptions, which often cannot be justified. On the other hand, it is not uncommon to combine analytical techniques with simulations, i.e., simulation is used not alone but in an interplay with analytical or semianalytical techniques.

1.3.1 Reasons for Simulation

There are a number of good reasons to perform simulations instead of performing experiments on real systems:

- Experiments are too *expensive*, too *dangerous*, or the system to be investigated does *not yet exist*. These are the main difficulties of experimentation with real systems, previously mentioned in Section 1.1.2, page 5.
- The *time scale* of the dynamics of the system is not compatible with that of the experimenter. For example, it takes millions of years to observe small changes in the development of the universe, whereas similar changes can be quickly observed in a computer simulation of the universe.
- Variables may be *inaccessible*. In a simulation all variables can be studied and controlled, even those that are inaccessible in the real system.

- Easy *manipulation* of models. Using simulation, it is easy to manipulate the parameters of a system model, even outside the feasible range of a particular physical system. For example, the mass of a body in a computer-based simulation model can be increased from 40 to 500 kg at a keystroke, whereas this change might be hard to realize in the physical system.
- Suppression of *disturbances*. In a simulation of a model it is possible to suppress disturbances that might be unavoidable in measurements of the real system. This can allow us to isolate particular effects and thereby gain a better understanding of those effects.
- Suppression of *second-order effects*. Often, simulations are performed since they allow suppression of second-order effects such as small nonlinearities or other details of certain system components, which can help us to better understand the primary effects.

1.3.2 Dangers of Simulation

The ease of use of simulation is also its most serious drawback: it is quite easy for the user to forget the limitations and conditions under which a simulation is valid, and therefore draw the wrong conclusions from the simulation. To reduce these dangers, one should always try to compare at least some results of simulating a model against experimental results from the real system. It also helps to be aware of the following three common sources of problems when using simulation:

- Falling in love with a model—the Pygmalion[2] effect. It is easy to become too enthusiastic about a model and forget all about the experimental frame, i.e., that the model is not the real world but only represents the real system under certain conditions. One example is the introduction of foxes on the Australian continent to solve the rabbit problem, on the model assumption that foxes hunt rabbits, which is true in many other parts of the world. Unfortunately, the foxes found the indigenous fauna much easier to hunt and largely ignored the rabbits.
- Forcing reality into the constraints of a model—the Procrustes[3] effect. One example is the shaping of our societies after currently fashionable economic theories having a simplified view of reality, and ignoring many other important aspects of human behavior, society, and nature.
- Forgetting the model's level of accuracy. All models have simplifying assumptions and we have to be aware of those in order to correctly interpret the results.

For these reasons, while analytical techniques are generally more restrictive since they have a much smaller domain of applicability, such techniques are more powerful when they apply. A simulation result is valid only for a particular set of input data. Many simulations are needed to gain an approximate understanding of a system. Therefore, if analytical techniques are applicable they should be used instead of simulation or as a complement.

1.4 Building Models

Given the usefulness of simulation in order to study the behavior of systems, how do we go about building models of those systems? This is the subject of most of this book and of the Modelica language, which has been created to simplify model construction as well as reuse of existing models.

There are in principle two main sources of general system-related knowledge needed for building mathematical models of systems:

[2] Pygmalion is a mythical king of Cyprus, who also was a sculptor. The king fell in love with one of his works, a sculpture of a young woman, and asked the gods to make her alive.
[3] Procrustes is a robber known from Greek mythology. He is known for the bed where he tortured travelers who fell into his hands: if the victim was too short, he stretched arms and legs until the person fit the length of the bed; if the victim was too tall, he cut off the head and part of the legs.

- The collected *general experience* in relevant domains of science and technology, found in the literature and available from experts in these areas. This includes the *laws of nature*, e.g. including Newton's laws for mechanical systems, Kirchhoff's laws for electrical systems, approximate relationships for nontechnical systems based on economic or sociological theories, etc., of
- The *system* itself, i.e., observations of and experiments on the system we want to model.

In addition to the above system knowledge, there is also specialized knowledge about mechanisms for handling and using facts in model construction for specific applications and domains, as well as generic mechanisms for handling facts and models, i.e.:

- *Application expertise*—mastering the application area and techniques for using all facts relative to a specific modeling application.
- *Software and knowledge engineering*—generic knowledge about defining, handling, using, and representing models and software, e.g. object orientation, component system techniques, expert system technology, etc.

What is then an appropriate analysis and synthesis *process* to be used in applying these information sources for constructing system models? Generally we first try to identify the main components of a system, and the kinds of interaction between these components. Each component is broken down into subcomponents until each part fits the description of an existing model from some model library, or we can use appropriate laws of nature or other relationships to describe the behavior of that component. Then we state the component interfaces and make a mathematical formulation of the interactions between the components of the model.

Certain components might have unknown or partially known model parameters and coefficients. These can often be found by fitting experimental measurement data from the real system to the mathematical model using *system identification*, which in simple cases reduces to basic techniques like curve fitting and regression analysis. However, advanced versions of system identification may even determine the form of the mathematical model selected from a set of basic model structures.

1.5 Analyzing Models

Simulation is one of the most common techniques for using models to answer questions about systems. However, there also exist other methods of analyzing models such as sensitivity analysis and model-based diagnosis, or analytical mathematical techniques in the restricted cases where solutions can be found in a closed analytical form.

1.5.1 Sensitivity Analysis

Sensitivity analysis deals with the question how *sensitive* the behavior of the model is to *changes* of model parameters. This is a very common question in design and analysis of systems. For example, even in well-specified application domains such as electrical systems, resistor values in a circuit are typically known only by an accuracy of 5 to 10 percent. If there is a large sensitivity in the results of simulations to small variations in model parameters, we should be very suspicious about the validity the model. In such cases small random variations in the model parameters can lead to large random variations in the behavior.

On the other hand, if the simulated behavior is not very sensitive to small variations in the model parameters, there is a good chance that the model fairly accurately reflects the behavior of the real system. Such robustness in behavior is a desirable property when designing new products, since they otherwise may become expensive to manufacture since certain tolerances must be kept very small.

However, there are also a number of examples of real systems which are very sensitive to variations of specific model parameters. In those cases that sensitivity should be reflected in models of those systems.

1.5.2 Model-Based Diagnosis

Model based *diagnosis* is a technique somewhat related to sensitivity analysis. We want to find the causes of certain behavior of a system by analyzing a model of that system. In many cases we want to find the causes of problematic and erroneous behavior. For example, consider a car, which is a complex system consisting of many interacting parts such as a motor, an ignition system, a transmission system, suspension, wheels, etc. Under a set of well-defined operating conditions each of these parts can be considered to exhibit a correct behavior if certain quantities are within specified value intervals. A measured or computed value outside such an interval might indicate an error in that component, or in another part influencing that component. This kind of analysis is called model-based diagnosis.

1.5.3 Model Verification and Validation

We have previously remarked about the dangers of simulation, e.g. when a model is not valid for a system regarding the intended simulation. How can we verify that the model is a good and reliable model, i.e., that it is valid for its intended use? This can be very hard, and sometimes we can hope only to get a partial answer to this question. However, the following techniques are useful to at least partially verify the validity of a model:

- Critically review the assumptions and approximations behind the model, including available information about the domain of validity regarding these assumptions.
- Compare simplified variants of the model to analytical solutions for special cases.
- Compare to experimental results for cases when this is possible.
- Perform sensitivity analysis of the model. If the simulation results are relatively insensitive to small variations of model parameters, we have stronger reasons to believe in the validity of the model.
- Perform internal consistency checking of the model, e.g. checking that dimensions or units are compatible across equations. For example, in Newton’s equation $F = m\ a$, the unit [N] on the left-hand side is consistent with [kg m s^{-2}] on the right-hand side.

In the last case it is possible for tools to automatically verify that dimensions are consistent if unit attributes are available for the quantities of the model. This functionality, however is yet not available for most current modeling tools.

1.6 Kinds of Mathematical Models

Different kinds of mathematical models can be characterized by different properties reflecting the behavior of the systems that are modeled. One important aspect is whether the model incorporates *dynamic* time-dependent properties or is *static*. Another dividing line is between models that evolve *continuously* over time, and those that change at *discrete* points in time. A third dividing line is between *quantitative* and *qualitative* models.

Certain models describe *physical distribution* of quantities, e.g. mass, whereas other models are *lumped* in the sense that the physically distributed quantity is approximated by being lumped together and represented by a single variable, e.g. a point mass.

Some phenomena in nature are conveniently described by stochastic processes and probability distributions, e.g. noisy radio transmissions or atomic-level quantum physics. Such models might be labeled *stochastic* or *probability*-based models where the behavior can be represented only in a statistic sense, whereas *deterministic* models allow the behavior to be represented without uncertainty. However, even stochastic models can be simulated in a "deterministic" way using a computer since the random number sequences often used to represent stochastic variables can be regenerated given the same seed values.

The same phenomenon can often be modeled as being either stochastic or deterministic depending on the level of detail at which it is studied. Certain aspects at one level are abstracted or averaged away at the next higher level. For example, consider the modeling of gases at different levels of detail starting at the quantum-mechanical elementary particle level, where the positions of particles are described by probability distributions:

- Elementary particles (orbitals)—stochastic models.
- Atoms (ideal gas model)—deterministic models.
- Atom groups (statistical mechanics)—stochastic models.
- Gas volumes (pressure and temperature)—deterministic models.
- Real gases (turbulence)—stochastic models.
- Ideal mixer (concentrations)—deterministic models.

It is interesting to note the kinds of model changes between stochastic or deterministic models that occur depending on what aspects we want to study. Detailed stochastic models can be averaged as deterministic models when approximated at the next upper macroscopic level in the hierarchy. On the other hand, stochastic behavior such as turbulence can be introduced at macroscopic levels as the result of chaotic phenomena caused by interacting deterministic parts.

1.6.1 Kinds of Equations

Mathematical models usually contain equations. There are basically four main kinds of equations, where we give one example of each.

Differential equations contain time derivatives such as $\frac{dx}{dt}$, usually denoted $\dot{x}$, e.g.:

$$\dot{x} = a \cdot x + 3 \tag{1-1}$$

Algebraic equations do not include any differentiated variables:

$$x^2 + y^2 = L^2 \tag{1-2}$$

Partial differential equations also contain derivatives with respect to other variables than time:

$$\frac{\partial a}{\partial t} = \frac{\partial^2 a}{\partial z^2} \tag{1-3}$$

Difference equations express relations between variables, e.g. at different points in time:

$$x(t+1) = 3x(t) + 2 \tag{1-4}$$

1.6.2 Dynamic vs. Static Models

All systems, both natural and man-made, are dynamic in the sense that they exist in the real world, which evolves in time. Mathematical models of such systems would be naturally viewed as *dynamic* in the sense that they evolve over time and therefore incorporate time. However, it is often useful to make the approximation of ignoring time dependence in a system. Such a system model is called *static*. Thus we can define the concepts of dynamic and static models as follows:

- A *dynamic* model includes *time* in the model. The word *dynamic* is derived from the Greek word *dynamis* meaning force and power, with dynamics being the (time-dependent) interplay between forces. Time can be included explicitly as a variable in a mathematical formula, or be present indirectly, e.g. through the time derivative of a variable or as events occurring at certain points in time.
- A *static* model can be defined *without* involving *time*, where the word *static* is derived from the Greek word *statikos,* meaning something that creates equilibrium. Static models are often used to describe systems in steady-state or equilibrium situations, where the output does not change if the input is the same. However, static models can display a rather dynamic behavior when fed with dynamic input signals.

It is usually the case that the behavior of a dynamic model is dependent on its *previous* simulation history. For example, the presence of a time derivative in mathematical model means that this derivative needs to be integrated to solve for the corresponding variable when the model is simulated, i.e., the *integration* operation takes the previous time history into account. This is the case, e.g. for models of capacitors where the voltage over the capacitor is proportional to the accumulated charge in the capacitor, i.e., integration/accumulation of the current through the capacitor. By differentiating that relation the time derivative of the capacitor voltage becomes proportional to the current through the capacitor. We can study the capacitor voltage increasing over time at a rate proportional to the current in Figure 1-3.

Another way for a model to be dependent on its previous history is to let preceding events influence the current state, e.g. as in a model of an ecological system where the number of prey animals in the system will be influenced by events such as the birth of predators. On the other hand, a dynamic model such as a sinusoidal signal generator can be modeled by a formula directly including time and not involving the previous time history.

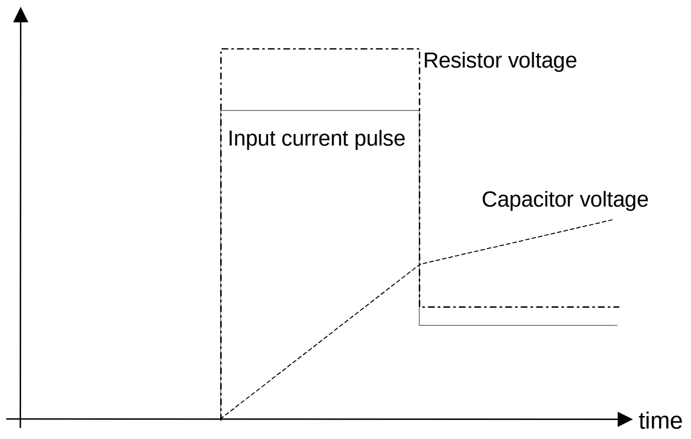


Figure 1-3. A resistor is a static system where the voltage is directly proportional to the current, independent of time, whereas a capacitor is a dynamic system where voltage is dependent on the previous time history.

A resistor is an example of a static model which can be formulated without including time. The resistor voltage is directly proportional to the current through the resistor, e.g. as depicted in Figure 1-3, with no dependence on time or on the previous history.

1.6.3 Continuous-Time vs. Discrete-Time Dynamic Models

There are two main classes of dynamic models: continuous-time and discrete-time models. The class of continuous-time models can be characterized as follows:

- *Continuous-time* models evolve their variable values continuously over time.

A variable from a continuous-time model A is depicted in Figure 1-4. The mathematical formulation of continuous-time models includes differential equations with time derivatives of some model variables. Many laws of nature, e.g. as expressed in physics, are formulated as differential equations.

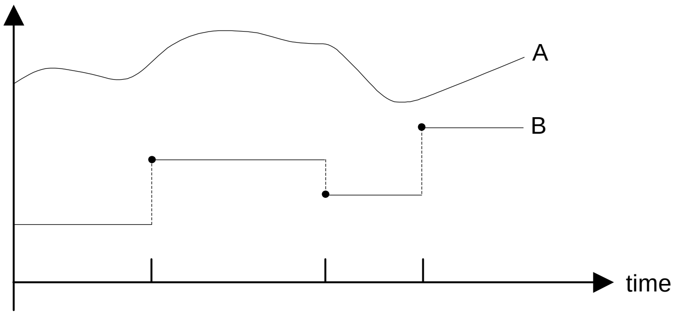


Figure 1-4. A discrete-time system B changes values only at certain points in time, whereas continuous-time systems like A evolve values continuously.

The second class of mathematical models is discrete-time models, e.g. as B in Figure 1-4, where variables change value only at certain points in time:

- *Discrete-time* models may change their variable values only at discrete points in time.

Discrete-time models are often represented by sets of difference equations, or as computer programs mapping the state of the model at one point in time to the state at the next point in time.

Discrete-time models occur frequently in engineering systems, especially computer-controlled systems. A common special case is sampled systems, where a continuous-time system is measured at regular time intervals and is approximated by a discrete-time model. Such sampled models usually interact with other discrete-time systems like computers. Discrete-time models may also occur naturally, e.g. an insect population which breeds during a short period once a year; i.e., the discretization period in that case is one year.

1.6.4 Quantitative vs. Qualitative Models

All of the different kinds of mathematical models previously discussed in this section are of a quantitative nature—variable values can be represented numerically according to a quantitatively measurable scale.

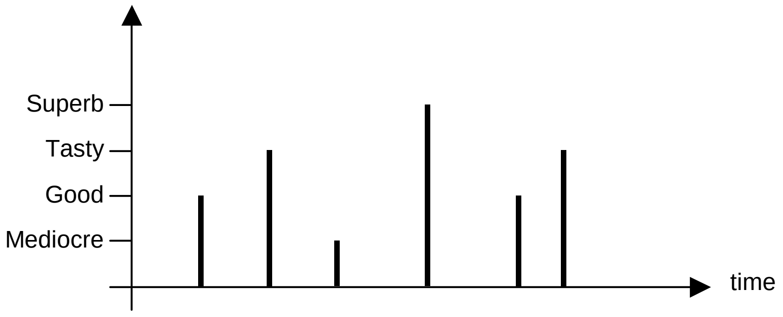


Figure 1-5. Quality of food in a restaurant according to inspections at irregular points in time.

Other models, so-called *qualitative* models, lack that kind of precision. The best we can hope for is a rough classification into a finite set of values, e.g. as in the food-quality model depicted in Figure 1-5. Qualitative models are by nature discrete-time, and the dependent variables are also discretized. However, even if the discrete values are represented by numbers in the computer (e.g. mediocre—1, good—2, tasty —3, superb—4), we have to be aware of the fact that the values of variables in certain qualitative models are not necessarily according to a linear measurable scale, i.e., tasty might not be three times better than mediocre.

1.7 Using Modeling and Simulation in Product Design

What role does modeling and simulation have in industrial product design and development? In fact, our previous discussion has already briefly touched this issue. Building mathematical models in the computer, so-called *virtual prototypes*, and simulating those models, is a way to quickly determine and optimize product properties without building costly physical prototypes. Such an approach can often drastically reduce development time and time-to-market, while increasing the quality of the designed product.

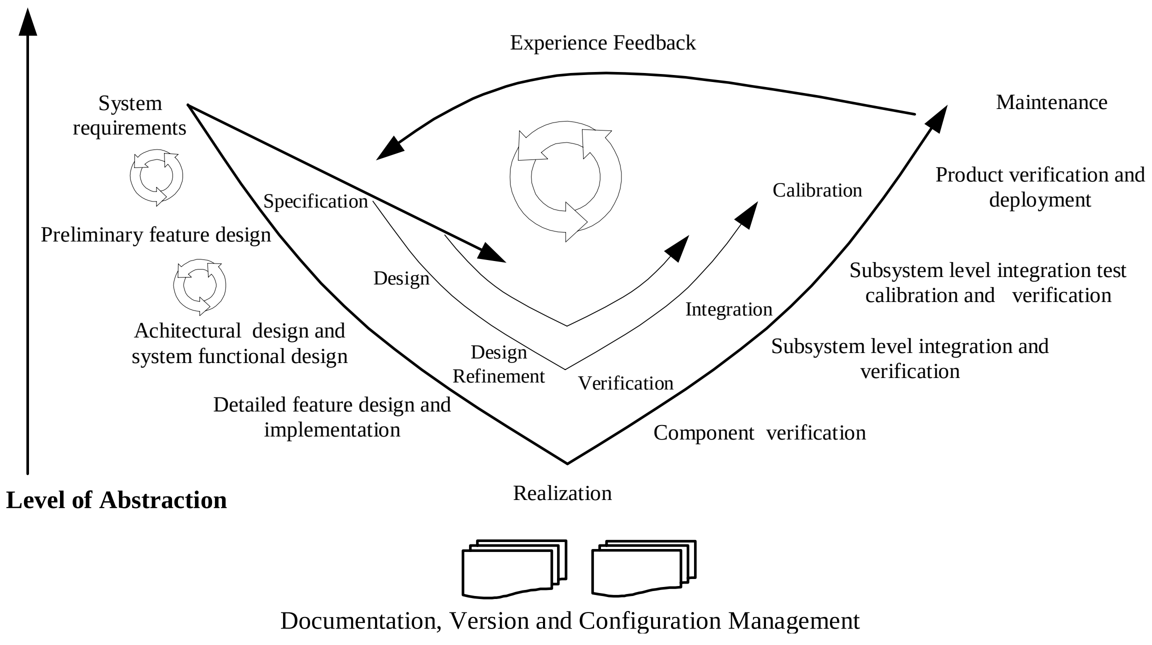

Experience Feedback
System requirements
Maintenance
Product verification and deployment
Calibration
Specification
Preliminary feature design
Design
Subsystem level integration test calibration and verification
Integration
Achitectural design and system functional design
Design Refinement
Verification
Subsystem level integration and verification
Detailed feature design and implementation
Component verification
Level of Abstraction
Realization
Documentation, Version and Configuration Management

Figure 1-6. The product design-V.

The so-called product *design-V*, depicted in Figure 1-6, includes all the standard phases of product development:

- Requirements analysis and specification.
- System design.
- Design refinement.
- Realization and implementation.
- Subsystem verification and validation.
- Integration.
- System calibration and model validation.
- Product deployment.

How does modeling and simulation fit into this design process?

In the first phase, *requirements analysis*, functional and nonfunctional requirements are specified. In this phase important design parameters are identified and requirements on their values are specified. For example, when designing a car there might be requirements on acceleration, fuel consumption, maximum emissions, etc. Those system parameters will also become parameters in our model of the designed product.

In the *system design phase* we specify the architecture of the system, i.e., the main components in the system and their interactions. If we have a simulation model component library at hand, we can use these library components in the design phase, or otherwise create new components that fit the designed product. This design process iteratively increases the level of detail in the design. A modeling tool that supports hierarchical system modeling and decomposition can help in handling system complexity.

The *implementation phase* will realize the product as a physical system and/or as a virtual prototype model in the computer. Here a virtual prototype can be realized before the physical prototype is built, usually for a small fraction of the cost.

In the *subsystem verification and validation phase*, the behavior of the subsystems of the product is verified. The subsystem virtual prototypes can be simulated in the computer and the models corrected if there are problems.

In the *integration phase* the subsystems are connected. Regarding a computer-based system model, the models of the subsystems are connected together in an appropriate way. The whole system can then be simulated, and certain design problems corrected based on the simulation results.

The system and model *calibration and validation phase* validates the model against measurements from appropriate physical prototypes. Design parameters are calibrated, and the design is often *optimized* to a certain extent according to what is specified in the original requirements.

During the last phase, *product deployment,* which usually only applies to the physical version of the product, the product is deployed and sent to the customer for feedback. In certain cases this can also be applied to virtual prototypes, which can be delivered and put in a computer that is interacting with the rest of the customer physical system in real time, i.e., hardware-in-the-loop simulation.

In most cases, experience feedback can be used to tune both models and physical products. All phases of the design process continuously interact with the model and design database, e.g. as depicted at the bottom of Figure 1-6.

1.8 Examples of System Models

In this section we briefly present examples of mathematical models from three different application areas, in order to illustrate the power of the Modelica mathematical modeling and simulation technology to be described in the rest of this book:

- A thermodynamic system—part of an industrial GTX100 gas turbine model.
- A 3D mechanical system with a hierarchical decomposition—an industry robot.
- A biochemical application —part of the citrate cycle (TCA cycle).

A connection diagram of the power cutoff mechanism of the GTX100 gas turbine is depicted in Figure 1-8 on page 16, whereas the gas turbine itself is shown in Figure 1-7 below.

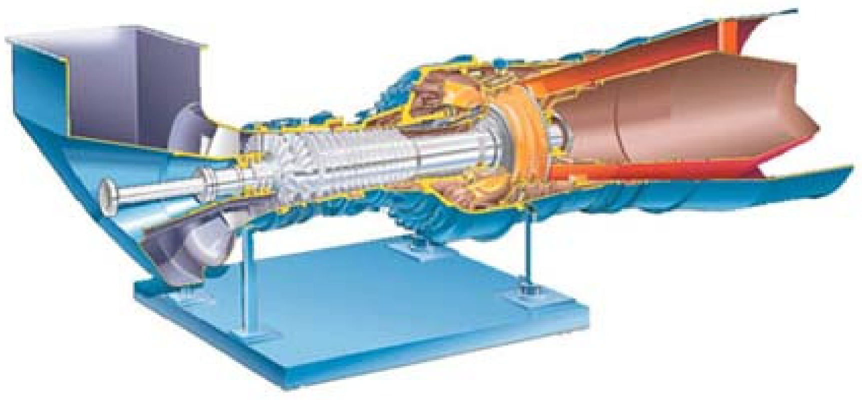


Figure 1-7. A schematic picture of the gas turbine GTX100. Courtesy Alstom Industrial Turbines AB, Finspång, Sweden.

This connection diagram might not appear as a mathematical model, but behind each icon in the diagram is a model component containing the equations that describe the behavior of the respective component.

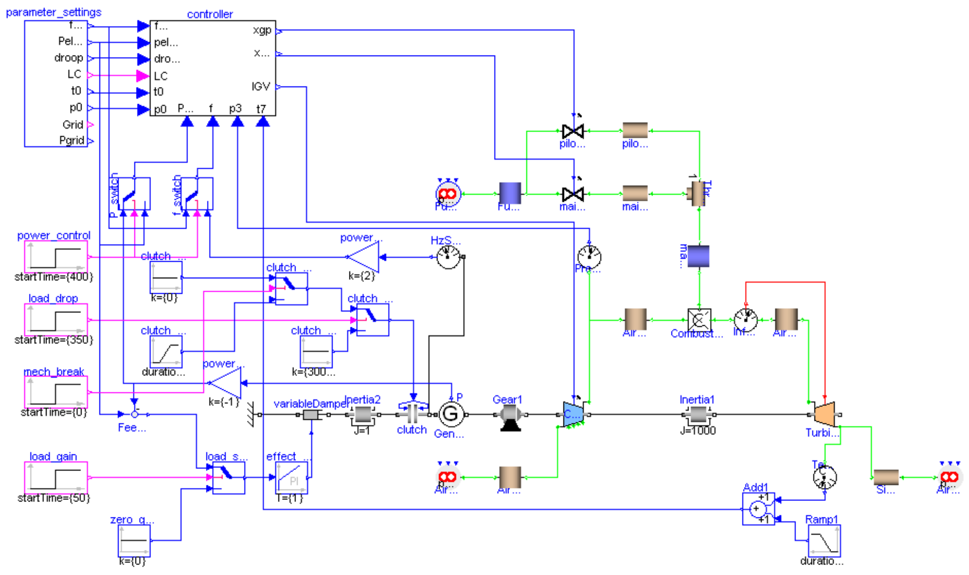


Figure 1-8. Detail of power cutoff mechanism in 40 MW GTX100 gas turbine model. Courtesy Alstom Industrial Turbines AB, Finspång, Sweden.

In Figure 1-9 we show a few plots from simulations of the gasturbine, which illustrates how a model can be used to investigate the properties of a given system.

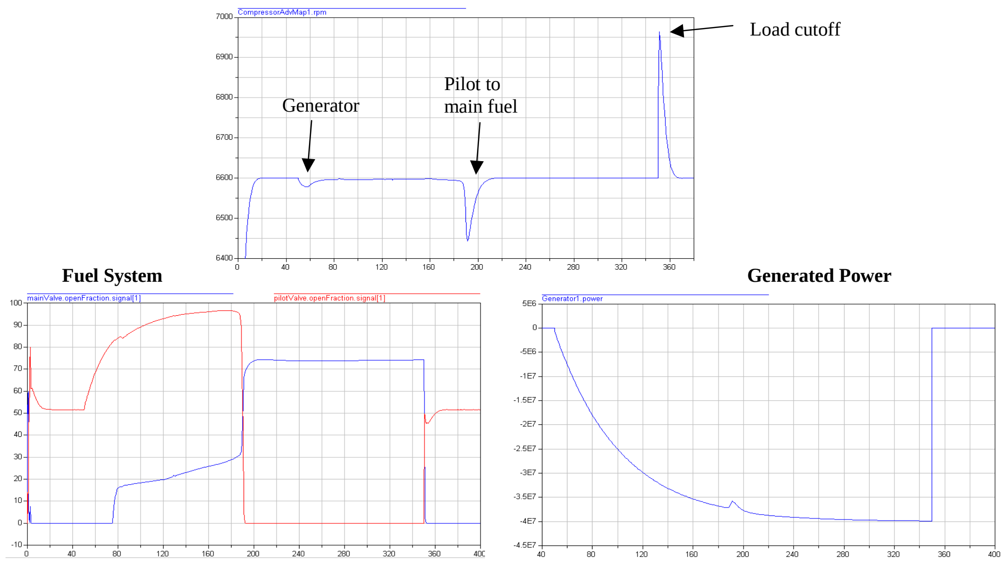
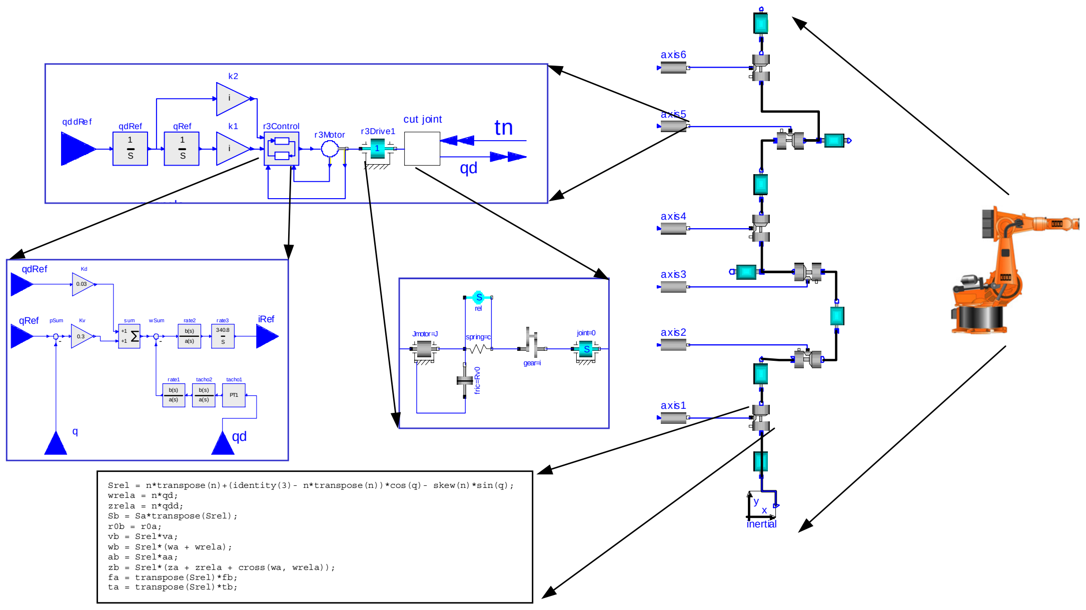


Figure 1-9. Simulation of GTX100 gas turbine power system cutoff mechanism. Courtesy Alstom Industrial Turbines AB, Finspång, Sweden

The second example, the industry robot, illustrates the power of hierarchical model decomposition. The three-dimensional robot, shown to the right of Figure 1-10, is represented by a 2D connection diagram (in the middle). Each part in the connection diagram can be a mechanical component such as a motor or joint, a control system for the robot etc. Components may consist of other components which can in turn can be decomposed. At the bottom of the hierarchy we have model classes containing the actual equations.



```
Srel = n*transpose(n)+(identity(3)- n*transpose(n))*cos(q)- skew(n)*sin(q);
wrela = n*qd;
zrela = n*qdd;
Sb = Sa*transpose(Srel);
r0b = r0a;
vb = Srel*va;
wb = Srel*(wa + wrela);
ab = Srel*aa;
zb = Srel*(za + zrela + cross(wa, wrela));
fa = transpose(Srel)*fb;
ta = transpose(Srel)*tb;
```

Figure 1-10. Hierarchical model of an industrial robot. Courtesy Martin Otter.

The third example is from an entirely different domain—biochemical pathways describing the reactions between reactants, in this particular case describing part of the citrate cycle (TCA cycle) as depicted in Figure 1-9.

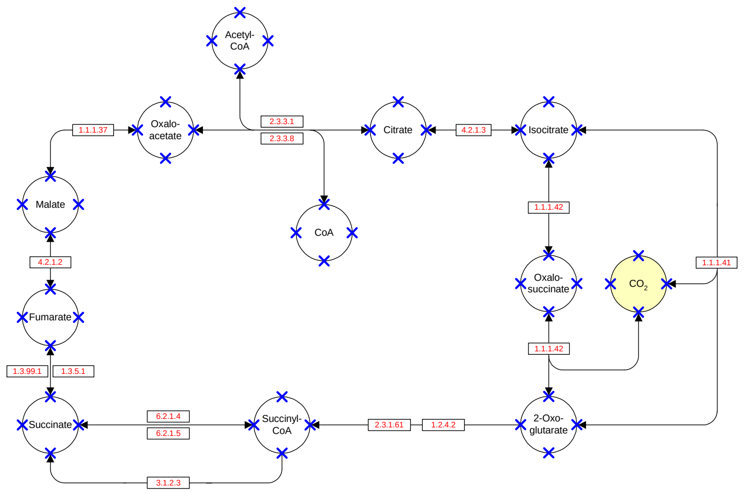


Figure 1-11. Biochemical pathway model of part of the citrate cycle (TCA cycle).

1.9 Summary

We have briefly presented important concepts such as system, model, experiment, and simulation. Systems can be represented by models, which can be subject to experiments, i.e., simulation. Certain models can be represented by mathematics, so-called mathematical models. This book is about object-oriented component-based technology for building and simulating such mathematical models. There are different classes of mathematical models, e.g. static versus dynamic models, continuous-time versus discrete-time models, etc., depending on the properties of the modeled system, the available information about the system, and the approximations made in the model.

1.10 Literature

Any book on modeling and simulation need to define fundamental concepts such as system, model, and experiment. The definitions in this chapter are generally available in modeling and simulation literature, including (Ljung and Glad 1994; Cellier 1991). The quote on system definitions, referring to the pendulum, is from (Ashby 1956). The example of different levels of details in mathematical models of gases presented in Section 1.6 is mentioned in (Hyötyniemi 2002). The product design-V process mentioned in Section 1.7 is described in (Stevens, Brook., Jackson, and Arnold 1998; Shumate and Keller 1992). The citrate cycle biochemical pathway part in Figure 1-11 is modeled after the description in (Michael 1999).

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:

```
simulate[5](HelloWorld,stopTime=2)
```

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

```
plot[6](x)
```

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

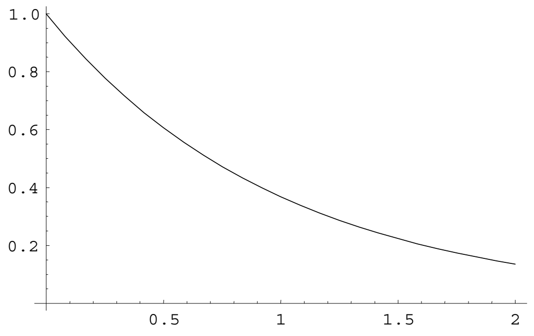


Figure 2-1. Plot of a simulation of the simple `HelloWorld` model.

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):

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

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

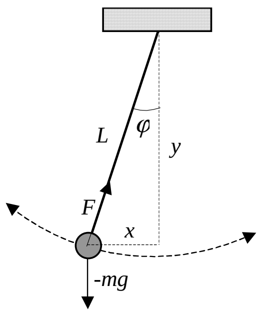


Figure 2-2. A planar pendulum.

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);
```

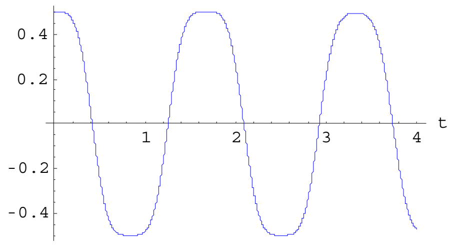


Figure 2-3. Plot of a simulation of the Pendulum DAE (differential algebraic equation) model.

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)
```

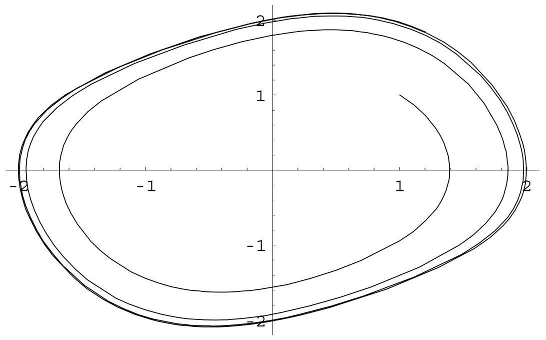


Figure 2-4. Parametric plot of a simulation of the Van der Pol oscillator model.

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).

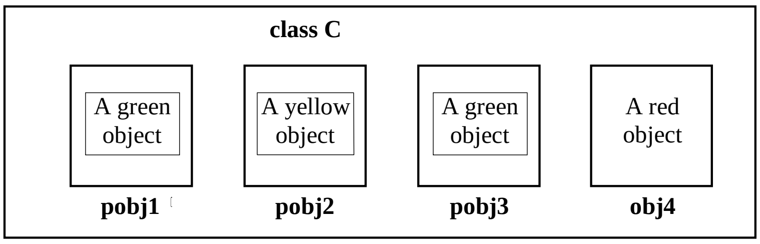


Figure 2-5. Three class parameters `pobj1`, `pobj2`, and `pobj3` that are instances (variables) of `class C`. These are essentially slots that can contain objects of different colors.

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`.

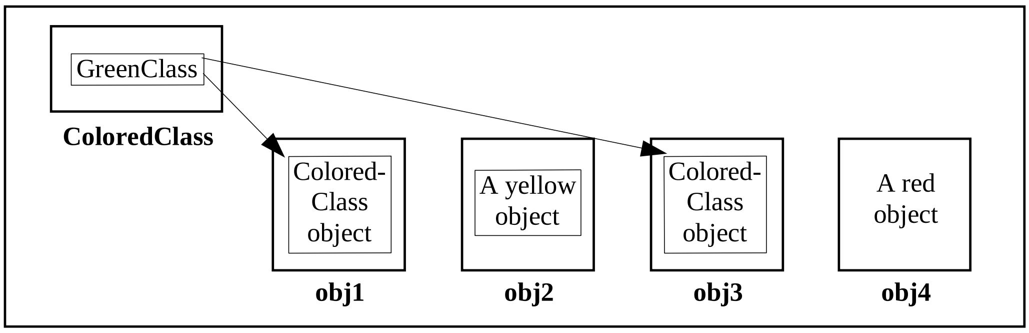


Figure 2-6. The class parameter `ColoredClass` is a type parameter that is propagated to the two member instance declarations of obj1 and obj3.

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.

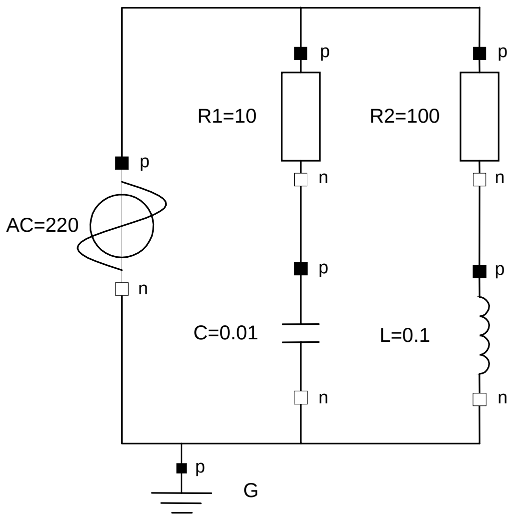


Figure 2-7. Connection diagram of the acausal simple circuit model.

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.

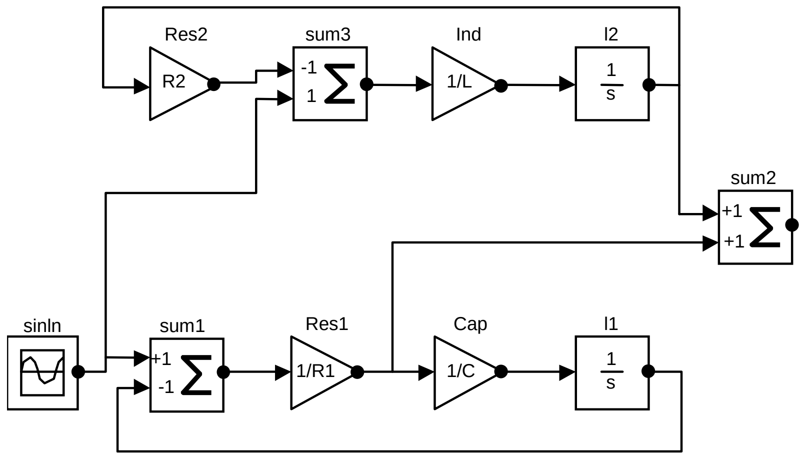


Figure 2-8 The simple circuit model using causal block-oriented modeling with explicit signal flow.

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.

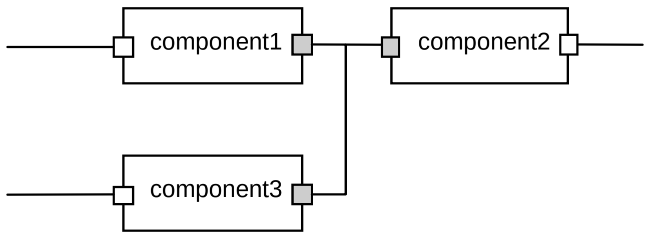


Figure 2-9. Schematic picture of a connection diagram for components.

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.

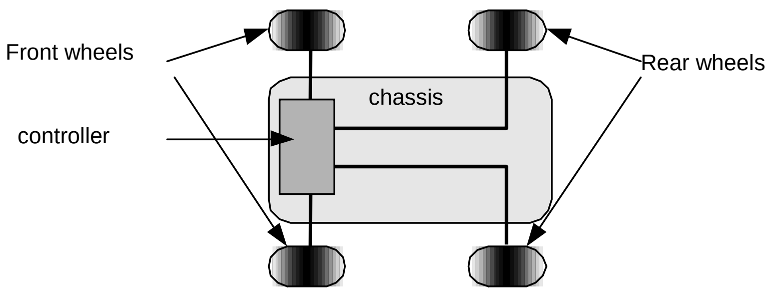


Figure 2-10. A connection diagram for a simple car model.

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");
```

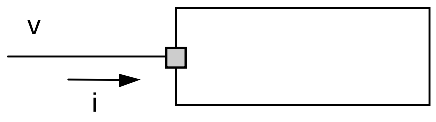


Figure 2-11. A component with one electrical `Pin` connector.

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.

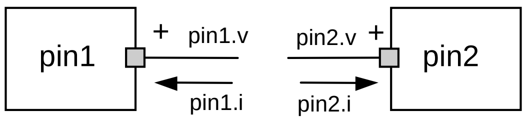


Figure 2-12. Connecting two components that have electrical pins.

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.

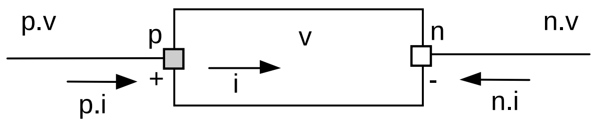


Figure 2-13. Generic `TwoPin` class that describes the general structure of simple electrical components with two pins.

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).

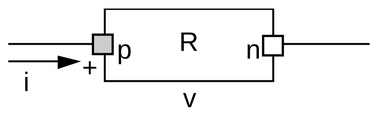


Figure 2-14. A resistor component.

```
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).

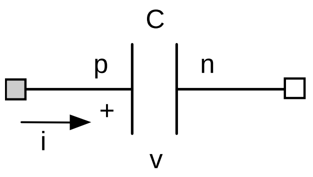


Figure 2-15. A capacitor component.

```
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

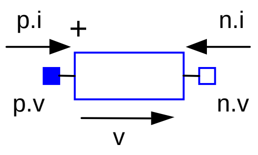


Figure 2-16. Resistor component.

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

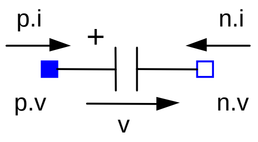


Figure 2-17. Capacitor component.

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

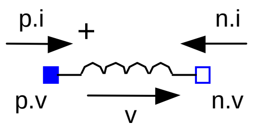


Figure 2-18. Inductor component.

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

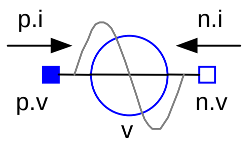


Figure 2-19. Voltage source component `VsourceAC`, where `v(t) = VA*sin(2*PI*f*time)`.

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

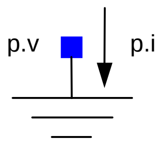


Figure 2-20. Ground component.

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.

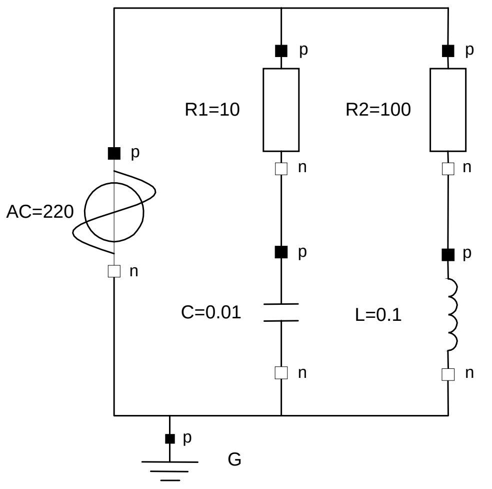


Figure 2-21. The simple circuit model.

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.

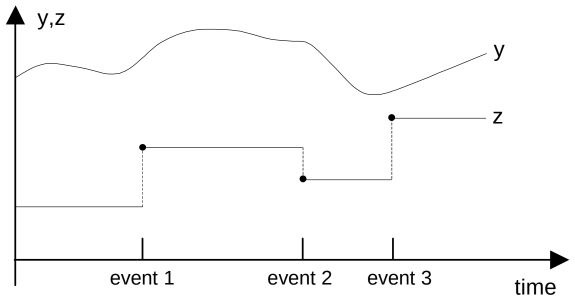


Figure 2-22. A discrete-time variable z changes value only at event instants, whereas continuous-time variables like y may change value both between and at events.

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.

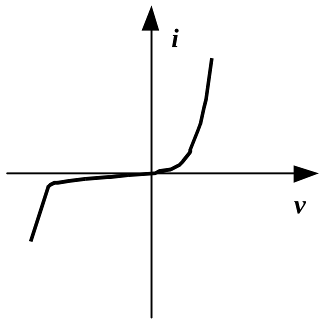


Figure 2-23. Real diode characteristic.

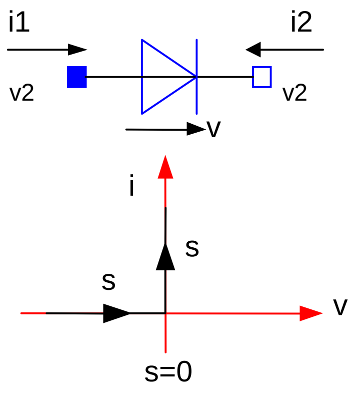


Figure 2-24. Ideal diode characteristic.

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.

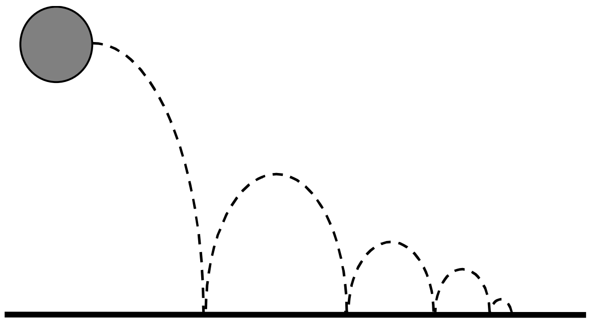


Figure 2-25. A bouncing ball.

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.

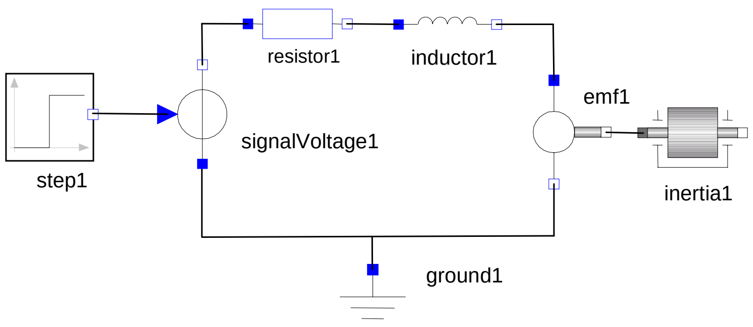
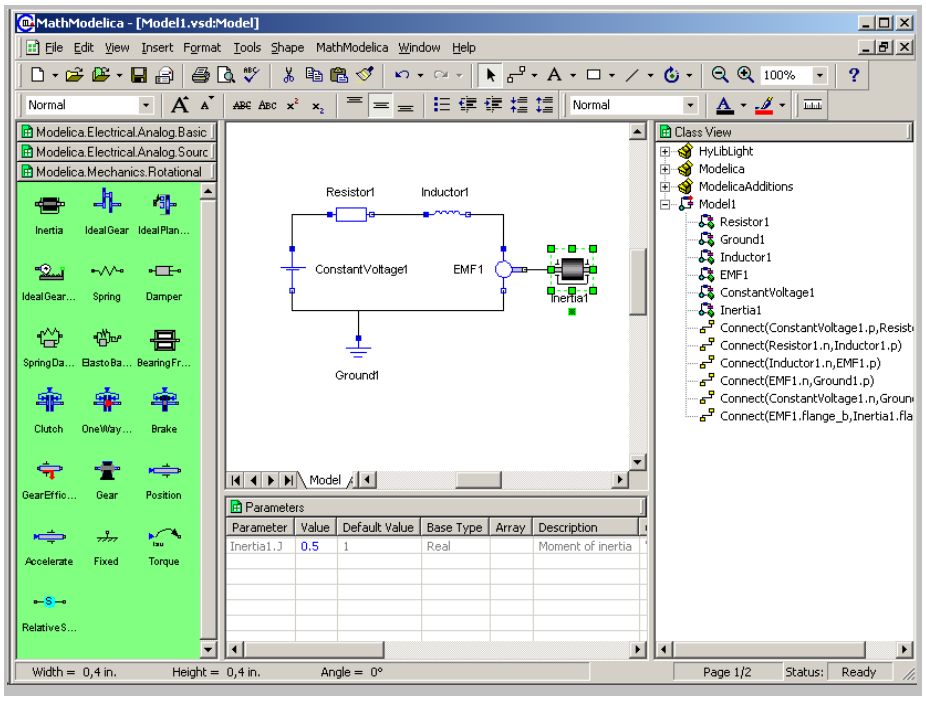


Figure 2-26. A multidomain `DCMotorCircuit` model with mechanical, electrical, and signal block components.

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.



Parameter	Value	Default Value	Base Type	Array	Description
Inertia1.J	0.5	1	Real		Moment of inertia

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 values of the variables during the specified simulation interval [t_0, t_1]. The result of the dynamic system simulation is a set of functions of time, such as `R2.v(t)` in the simple circuit model. Those functions can be displayed as graphs and/or saved in a file.

In most cases (but not always) the performance of generated simulation code (including the solver) is similar to handwritten C code. Often Modelica is more efficient than straightforwardly written C code, because additional opportunities for symbolic optimization are used by the system, compared to what a human programmer can manually handle.

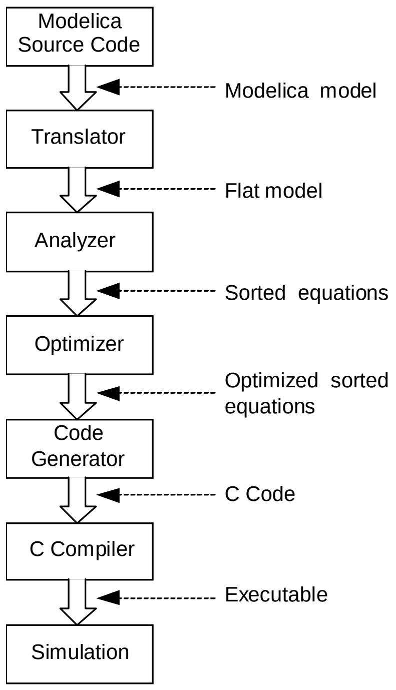


Figure 2-28. The stages of translating and executing a Modelica model.

2.20.1 Hand Translation of the Simple Circuit Model

Let us return once more to the simple circuit model, previously depicted in Figure 2-7, but for the reader's convenience also shown below in Figure 2-29. It is instructive to translate this model by hand, in order to understand the process.

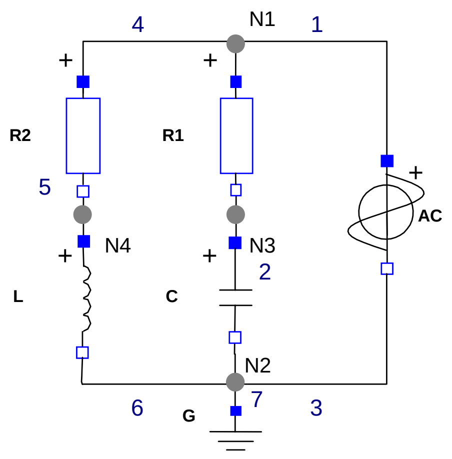


Figure 2-29. The `SimpleCircuit` model once more, with explicitly labeled connection nodes N1, N2, N3, N4, and wires 1 to 7.

Classes, instances and equations are translated into a flat set of equations, constants, and variables (see the equations in Table 2-1), according to the following rules:

1. For each class instance, add one copy of all equations of this instance to the total differential algebraic equation (DAE) system or ordinary differential equation system (ODE)—both alternatives can be possible, since a DAE in a number of cases can be transformed into an ODE.
2. For each connection between instances within the model, add connection equations to the DAE system so that nonflow variables are set equal and flow variables are summed to zero.

The equation `v=p.v-n.v` is defined by the class `TwoPin`. The `Resistor` class inherits the `TwoPin` class, including this equation. The `SimpleCircuit` class contains a variable `R1` of type `Resistor`. Therefore, we include this equation instantiated for `R1` as `R1.v=R1.p.v-R1.n.v` into the system of equations.

The wire labeled `1` is represented in the model as `connect(AC.p, R1.p)`. The variables `AC.p` and `R1.p` have type `Pin`. The variable `v` is a *nonflow* variable representing voltage potential. Therefore, the equality equation `R1.p.v=AC.p.v` is generated. Equality equations are always generated when nonflow variables are connected.

Notice that another wire (labeled `4`) is attached to the same pin, `R1.p`. This is represented by an additional connect equation: `connect(R1.p.R2.p)`. The variable `i` is declared as a flow variable. Thus, the equation `AC.p.i+R1.p.i+R2.p.i=0` is generated. Zero-sum equations are always generated when connecting flow variables, corresponding to Kirchhoff's second law.

The complete set of equations (see Table 2-1) generated from the `SimpleCircuit` class consists of 32 differential-algebraic equations. These include 32 variables, as well as `time` and several parameters and constants.

Table 2-1. The equations extracted from the simple circuit model—an implicit DAE system.

AC	`0 = AC.p.i+AC.n.i`
	`AC.v = Ac.p.v-AC.n.v`
	`AC.i = AC.p.i`
	`AC.v = AC.VA*`
	`sin(2*AC.PI*`
	`AC.f*time);`
		L	`0 = L.p.i+L.n.i`
			`L.v = L.p.v-L.n.v`
			`L.i = L.p.i`
			`L.v = L.L*der(L.i)`

R1	`0 = R1.p.i+R1.n.i`
	`R1.v = R1.p.v-R1.n.v`
	`R1.i = R1.p.i`
	`R1.v = R1.R*R1.i`
		G	`G.p.v = 0`

R2	`0 = R2.p.i+R2.n.i`
	`R2.v = R2.p.v-R2.n.v`
	`R2.i = R2.p.i`
	`R2.v = R2.R*R2.i`
		wires	`R1.p.v = AC.p.v // wire 1`
			`C.p.v = R1.n.v // wire 2`
			`AC.n.v = C.n.v // wire 3`
			`R2.p.v = R1.p.v // wire 4`
			`L.p.v = R2.n.v // wire 5`
			`L.n.v = C.n.v // wire 6`
			`G.p.v = AC.n.v // wire 7`

C	`0 = C.p.i+C.n.i`
	`C.v = C.p.v-C.n.v`
	`C.i = C.p.i`
	`C.i = C.C*der(C.v)`
		flow at node	`0 = AC.p.i+R1.p.i+R2.p.i // N1`
			`0 = C.n.i+G.p.i+AC.n.i+L.n.i // N2`
			`0 = R1.n.i + C.p.i // N3`
			`0 = R2.n.i + L.p.i // N4`

Table 2-2 gives the 32 variables in the system of equations, of which 30 are algebraic variables since their derivatives do not appear. Two variables, `C.v` and `L.i`, are dynamic variables since their derivatives occur in the equations. In this simple example the dynamic variables are state variables, since the DAE reduces to an ODE (also see Section 18.2.6, page 686).

Table 2-2. The variables extracted from the simple circuit model.

`R1.p.i`	`R1.n.i`	`R1.p.v`	`R1.n.v`	`R1.v`
`R1.i`	`R2.p.i`	`R2.n.i`	`R2.p.v`	`R2.n.v`
`R2.v`	`R2.i`	`C.p.i`	`C.n.i`	`C.p.v`
`C.n.v`	`C.v`	`C.i`	`L.p.i`	`L.n.i`
`L.p.v`	`L.n.v`	`L.v`	`L.i`	`AC.p.i`
`AC.n.i`	`AC.p.v`	`AC.n.v`	`AC.v`	`AC.i`
`G.p.i`	`G.p.v`			

2.20.2 Transformation to State Space Form

The implicit differential algebraic system of equations (DAE system) in Table 2-1 should be further transformed and simplified before applying a numerical solver. The next step is to identify the kind of variables in the DAE system. We have the following four groups:

1. All constant variables which are model parameters, thus easily modified between simulation runs and declared with the prefixed keyword `parameter`, are collected into a parameter vector p. All other constants can be replaced by their values, thus disappearing as named constants.
2. Variables declared with the input attribute, i.e., prefixed by the `input` keyword, that appears in instances at the highest hierarchical level, are collected into an input vector *u*.
3. Variables whose derivatives appear in the model (dynamic variables), i.e., the `der`( ) operator is applied to those variables, are collected into a state vector *x*, (for exceptions see Section 18.2.6, page 686).
4. All other variables are collected into a vector *y* of algebraic variables, i.e., their derivatives do not appear in the model.

For our simple circuit model these four groups of variables are the following:

p = {`R1.R, R2.R, C.C, L.L, AC.VA, AC.f`}
u = {`AC.v`}
x = {`C.v,L.i`}
y = {`R1.p.i, R1.n.i, R1.p.v, R1.n.v, R1.v, R1.i, R2.p.i, R2.n.i, R2.p.v, R2.n.v, R2.v, R2.i, C.p.i, C.n.i, C.p.v, C.n.v, C.i, L.n.i, L.p.v, L.n.v, L.v, AC.p.i, AC.n.i, AC.p.v, AC.n.v, AC.i, AC.v, G.p.i, G.p.v`}

We would like to express the problem as the smallest possible ordinary differential equation (ODE) system (in the general case a DAE system) and compute the values of all other variables from the solution of this minimal problem. The system of equations should preferably be in explicit state space form as below.

$$\dot{x} = f(x,t) \tag{2-3}$$

That is, the derivative $\dot{x}$ with respect to time of the state vector *x* is equal to a function of the state vector *x* and time. Using an iterative numerical solution method for this ordinary differential equation system, at each iteration step, the derivative of the state vector is computed from the state vector at the current point in time.

For the simple circuit model we have the following:

x = {`C.v,L.i`}, *u* = {`AC.v`} (with constants:
`R1.R,R2.R,C.C,L.L,` (2-4)
$\dot{x}$ = {**`der`**`(C.v)`, **`der`**`(L.i)`} `AC.VA,AC.f,AC.PI`)

2.20.3 Solution Method

We will use an iterative numerical solution method. First, assume that an estimated value of the state vector *x* ={`C.v,L.i`} is available at `t=0` when the simulation starts. Use a numerical approximation for the derivative $\dot{x}$ (i.e. `der(x)`) at time `t`, e.g.:

```
der(x) = (x(t+h)-x(t))/h                                        (2-5)
```

giving an approximation of *x* at time t+h:

```
x(t+h) = x(t) + der(x)*h                                        (2-6)
```

In this way the value of the state vector *x* is computed one step ahead in time for each iteration, provided `der(x)` can be computed at the current point in simulated time. However, the derivative `der(x)` of the state vector can be computed from $\dot{x} = f(x,t)$, i.e., by selecting the equations involving `der(x)`, and algebraically extracting the variables in the vector *x* in terms of other variables, as below:

```
der(C.v) = C.i/C.C
der(L.i) = L.v/L.L                                              (2-7)
```

Other equations in the DAE system are needed to calculate the unknowns `C.i` and `L.v` in the above equations. Starting with `C.i`, using a number of different equations together with simple substitutions and algebraic manipulations, we derive equations (2-8) through (2-10) below.

```
C.i = R1.v/R1.R                                                 (2-8)
using: C.i = C.p.i = -R1.n.i = R1.p.i = R1.i = R1.v/R1.R


R1.v = R1.p.v - R1.n.v = R1.p.v – C.v
using: R1.n.v = C.p.v = C.v+C.n.v = C.v + AC.n.v = C.v + G.p.v =   (2-9)
              = C.v + 0 = C.v


R1.p.v  = AC.p.v = AC.VA*sin(2*AC.f*AC.PI*t)
using: AC.p.v = AC.v+AC.n.v = AC.v+G.p.v =                      (2-10)
              = AC.VA*sin(2*AC.f*AC.PI*t)+0
```

In a similar fashion we derive equations (2-11) and (2-12) below:

```
L.v  = L.p.v - L.n.v = R1.p.v - R2.v
using: L.p.v = R2.n.v = R1.p.v-R2.v                             (2-11)
and: L.n.v = C.n.v = AC.n.v = G.p.v = 0


R2.v = R2.R*L.p.i
using: R2.v  =  R2.R*R2.i  =  R2.R*R2.p.i  =  R2.R*(-R2.n.i)  =   (2-12)
R2.R*L.p.i = R2.R*L.i
```

Collecting the five equations together:

```
C.i     = R1.v/R1.R
R1.v    = R1.p.v - C.v
R1.p.v  = AC.VA*sin(2*AC.f*AC.PI*time)                         (2-13)
L.v     = R1.p.v - R2.v
R2.v    = R2.R*L.i
```

By sorting the equations in data-dependency order, and converting the equations to assignment statements—this is possible since all variable values can now be computed in order—we arrive at the following set of assignment statements to be computed at each iteration, given values of `C.v`, `L.i`, and `t` at the same iteration:

```
R2.v       := R2.R*L.i
R1.p.v     := AC.VA*sin(2*AC.f*AC.PI*time)
L.v        := R1.p.v - R2.v
R1.v       := R1.p.v - C.v
C.i        := R1.v/R1.R
der(L.i)   := L.v/L.L
der(C.v)   := C.i/C.C
```

These assignment statements can be subsequently converted to code in some programming language, e.g., C, and executed together with an appropriate ODE solver, usually using better approximations to derivatives and more sophisticated forward-stepping schemes than the simple method described above, which, by the way, is called the *Euler integration* method. The algebraic transformations and sorting procedure that we somewhat painfully performed by hand on the simple circuit example can be performed completely automatically, and is known as *BLT-transformation*, i.e., conversion of the equation system coefficient matrix into block lower triangular form (Figure 2-30).

	R2.v	R1.p.v	L.v	R1.v	C.i	L.i	C.v
`R2.v = R2.R*L.i`	1	0	0	0	0	0	0
`R1.p.v = AC.VA*sin(2*AC.f*AC.PI*time)`	0	1	0	0	0	0	0
`L.v = R1.p.v - R2.v`	1	1	1	0	0	0	0
`R1.v = R1.p.v - C.v`	0	1	0	1	0	0	0
`C.i = R1.v/R1.R`	0	0	0	0	1	0	0
`der(L.i) = L.v/L.L`	0	0	1	0	0	1	0
`der(C.v) = C.i/C.C`	0	0	0	0	1	0	1

Figure 2-30. Block lower triangular form of the `SimpleCircuit` example.

The remaining 26 algebraic variables in the equation system of the simple circuit model that are not part of the minimal 7-variable kernel ODE system solved above can be computed at leisure for those iterations where their values are desired—this is not necessary for solving the kernel ODE system.

It should be emphasized that the simple circuit example is trivial. Realistic simulation models often contain tens of thousands of equations, nonlinear equations, hybrid models, etc. The symbolic transformations and reductions of equation systems performed by a real Modelica compiler are much more complicated than what has been shown in this example, e.g., including index reduction of equations and tearing of subsystems of equations.

```
simulate(SimpleCircuit,stopTime=5)]
plot(C.v, xrange={0,5})
```

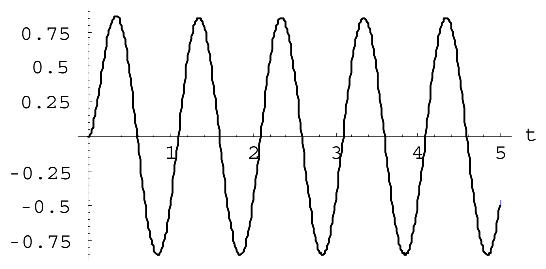


Figure 2-31. Simulation of the `SimpleCircuit` model with a plot of the voltage `C.v` over the capacitor.

2.21 History

In September 1996, a group of tool designers, application experts, and computer scientists joined forces to work together in the area of object-oriented modeling technology and applications. The group included specialists behind the object-oriented modeling languages Dymola, Omola, ObjectMath, NMF (Neutral Modeling Format), Allan-U.M.L, SIDOPS+, and Smile, even though not all were able to attend the first meeting. Initially the goal was to write a white paper on object-oriented modeling language technology including possible unification of existing modeling languages, as part of an action in the ESPRIT project Simulation in Europe Basic Research Working Group (SiE-WG).

However, the work quickly focused on the more ambitious goal of creating a new, unified object-oriented modeling language based on the experience from the previous designs. The designers made an effort to unify the concepts to create a common language, starting the design from scratch. This new language is called *Modelica*.

The group soon established itself as Technical Committee 1 within EuroSim, and as the Technical Chapter on Modelica within Society for Computer Simulation International (SCS). In February 2000, Modelica Association was formed as an independent nonprofit international organization for supporting and promoting the development and dissemination of the Modelica Language and the Modelica Standard Libraries.

The first Modelica language description, version 1.0, was put on the web in September 1997, by a group of proud designers, after a number of intense design meetings. Today, in 2003, at the date of this writing, the Modelica 2.1 specification has been released—the result of a large amount of work including 34 three-day design meetings. Two complete commercial tools supporting textual and graphical model design and simulation with Modelica are currently available, as well as a rather complete open-source implementation, and several partial university prototypes. A large and growing Modelica Standard Library is also available. The language is quickly spreading both in industry and in academia.

If we trace back a few steps and think about the Modelica technology, two important points become apparent:

- Modelica includes *equations*, which is unusual in most programming languages.
- The Modelica technology includes *graphical* editing for application model design based on predefined components.

In fact, concerning the first point, equations were used very early in human history—already in the third millennium B.C. At that point the well-known equality sign for equations was not yet invented. That happened much later—the equation sign was introduced by Robert Recorde in 1557, in a form depicted in Figure 2-32.

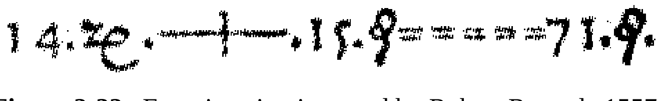
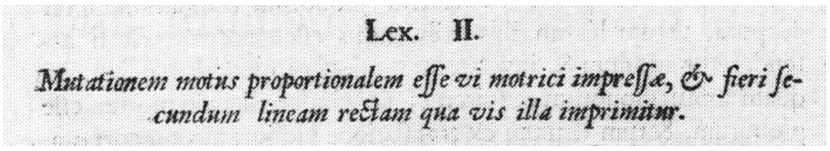


Figure 2-32. Equation sign invented by Robert Recorde 1557. Reproduced from Figure 4.1-1 on page 81 in (Gottwald, Gellert, Kuestner 1989), courtesy Thompson Inc.

However, it took a while for this invention to spread in Europe. Even a hundred years later, Newton (in his *Principia*, vol. 1, 1686) still wrote his famous law of motion as text in Latin, as shown in Figure 2-33. Translated to English this can be expressed as: “The change of motion is proportional to the motive force impressed”.


Lex. II.

Mutationem motus proportionalem eſſe vi motrici impreſſæ, & fieri ſecundum lineam rectam qua vis illa imprimitur.

Figure 2-33. Newton's famous second law of motion in Latin. Translated to English this becomes "The change of motion is proportional to the motive force impressed". Reproduced from Figure "Newton's laws of motion" on page 51 in (Fauvel, Flood, Shortland, and Wilson 1990), courtesy Oxford University Press.

In modern mathematical syntax, Newton's law of motion appears as follows:

$$\frac{d}{dt}(m \cdot v) = \sum_i F_i \qquad (2\text{-}14)$$

This is an example of a differential equation. The first simulators to solve such equations were analog. The idea is to model the system in terms of ordinary differential equations and then make a physical device that obeys the equations. The first analog simulators were mechanical devices, but from the 1950s electronic analog simulators became predominant. A number of electronic building blocks such as adders, multipliers, integrators, etc. could be interconnected by cables as depicted in Figure 2-34.

Concerning the further development, for a long time equations were quite rare in computer languages. Early versions of Lisp systems and computer algebra systems were being developed, but mostly for formula manipulation rather than for direct simulation.

However, a number of simulation languages for digital computers soon started to appear. The first equation-based modeling tool was Speed-Up, a package for chemical engineering and design introduced in 1964. Somewhat later, in 1967, Simula 67 appeared—the first object-oriented programming language, with profound influence on programming languages and somewhat later on modeling languages. The same year the CSSL (Continuous System Simulation Language) report unified existing notations for expressing continuous system simulation models and introduced a common form of causal "equation", e.g.:

$$\begin{aligned} &\textit{variable} = \textit{expression} \\ &v = \mathrm{INTEG}(F)\,/\,m \end{aligned} \qquad (2\text{-}15)$$

The second equation is a variant of the equation of motion: the velocity is the integral of the force divided by the mass. These are not general equations in the mathematical sense since the causality is from right to left, i.e., an equation of the form *expression* = *variable* is not allowed. However, it is still a definite step forward toward a more general executable computer representation of equation-based mathematical models. ACSL, first introduced in 1976, was a rather common simulation system initially based on the CSSL standard.

An important pioneering predecessor to Modelica is Dymola (*Dynamic modeling language*—not today's Dymola tool meaning Dynamic modeling laboratory) described in Hilding Elmqvist's PhD thesis 1978. This was the first work to recognize the importance of modeling with acausal equations together with hierarchical submodels and methods for automatic symbolic manipulation to support equation solution. The GASP-IV system in 1974 followed by GASP-V a few years later introduced integrated continuous-discrete simulation. The Omola language (1989) is a modeling language with full object orientation including inheritance as well as hybrid simulation. The Dymola language was later (1993) enhanced with inheritance, as well as with mechanisms for discrete-event handling and more efficient symbolic-numeric equation system solution methods.

Other early object-oriented acausal modeling languages include NMF (Natural Model Format, 1989), primarily used for building simulation, Allan-U.M.L, SIDOPS+ supporting bond graph modeling, and Smile (1995)—influenced by Objective C. Two other important languages that should be mentioned are ASCEND (1991) and gPROMS (1994).

This author’s acquaintance with equation-based modeling and problem solving started in 1975 by solving the Schrödinger equation for a very specific case in solid-state physics, using the pseudopotential approximation. Later, in 1989, I initiated development of a new object-oriented modeling language called ObjectMath together with my brother Dag Fritzson. This was one of the earlier object-oriented computer algebra and simulation systems, integrated with Mathematica and with a general parameterized generic class concept, as well as code generation to C++ for efficient simulation of industrial applications. The fourth version of ObjectMath was completed in the fall of 1996 when I decided to join the Modelica effort instead of continuing with a fifth version of ObjectMath.

Concerning the second aspect mentioned earlier—graphical specification of simulation models—Figure 2-34 is tells an interesting story.

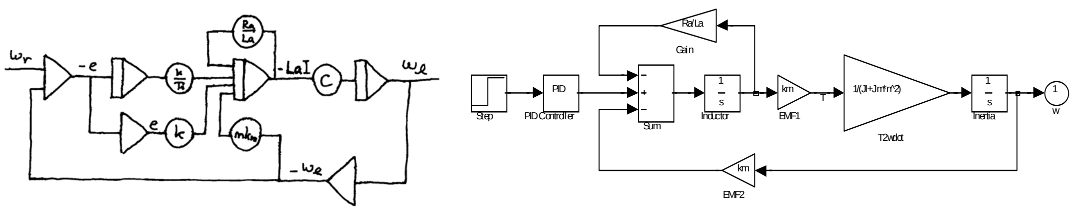


Figure 2-34. Analog computing vs. graphical block diagram modeling on modern digital computers. Courtesy Karl-Johan Åström and Hilding Elmqvist.

The left part of the figure shows the circuitry of an analog computer with its building blocks connected by cables. The right figure is a block diagram of very similar structure, directly inspired by the analog computing paradigm but executed on digital computers. Such block diagrams are typically constructed by common tools available today such as Simulink or SystemBuild. Block diagrams represent causal equations since there is a specified data-flow direction.

The connection diagrams used in Modelica graphical modeling include connections between instances of classes containing acausal equations, as first explored in the Hibliz system. This is a generalization inspired by the causal analog computing circuit diagrams and block diagrams. The Modelica connection diagrams have the advantage of supporting natural physical modeling since the topology of a connection diagram directly corresponds to the structure and decomposition of the modeled physical system.

2.22 Summary

This chapter has given a quick overview of the most important concepts and language constructs in Modelica. However, Chapter 3 to Chapter 11 together with Chapter 13 give a much more complete description of the language with many examples including a number of language and usage aspects not covered by this quick tour. We have also defined important concepts such as object oriented mathematical modeling and acausal physical modeling, and briefly presented the concepts and Modelica language constructs for defining components, connections, and connectors. The chapter concludes with an in-depth example of the translation and execution of a simple model, and a short history of equations and mathematical modeling languages up to and including Modelica from ancient times until today.

2.23 Literature

Many programming language books are organized according to a fairly well-established pattern of first presenting a quick overview of the language, followed by a more detailed presentation according to the most important language concepts and syntactic structures. This book is no exception to that rule, where this chapter constitutes the quick overview. As in many other texts we start with a `HelloWorld` example, e.g as in the Java programming language book (Arnold and Gosling 1999), but with a different contents since printing an “Hello World” message is not very relevant for an equation-based language.

The most important reference document for this chapter is the Modelica tutorial (Modelica Association 2000), of which the first version including a design rationale (Modelica Association 1997) was edited primarily by Hilding Elmqvist. Several examples, code fragments, and text fragments in this chapter are based on similar ones in the tutorial, e.g. the `SimpleCircuit` model with the simple electrical components, the `polynomialEvaluator`, the low pass filter, the ideal `Diode`, the `BouncingBall` model, etc. Figure 2-8 on block oriented modeling is also from the tutorial.

The hand translation of the simple circuit model is inspired by a similar but less elaborated example in a series of articles by Martin Otter *et al* (Otter 1999). The recent history of mathematical modeling languages is described in some detail in (Åström, Elmqvist, and Mattsson 1998), whereas bits and pieces of the ancient human history of the invention and use of equations can be found in (Gottwald, Gellert, and Kuestner 1989), and the picture on Newton’s second law in latin in (Fauvel, Flood, Shortland, and Wilson 1990). Early work on combined continous/discrete simulation is described in (Pritsker 1974) followed by (Cellier 1979). This author’s first simulation work involving solution of the Schrödinger equation for a particular case is described in (Fritzson and Berggren 1976).

Current versions of several Modelica tools are described in Chapter 19, including OpenModelica, MathModelica, and Dymola meaning the Dynamic Modeling Laboratory. The predecessors of the Modelica language are briefly described in Appendix F, including Dymola meaning the Dynamic Modeling Language: (Elmqvist 1978; Elmqvist, Brück, and Otter 1996), Omola: (Mattsson, Andersson, and Åström 1993; Andersson 1994), ObjectMath: (Fritzson, Viklund, Herber, Fritzson 1992), (Fritzson, Viklund, Fritzson, and Herber 1995; Viklund and Fritzson 1995), NMF: (Sahlin, Bring, Sowell 1996), Smile: (Ernst, Jähnichen, and Klose 1997), etc.

Speed-Up, the earliest equation-based simulation tool, is presented in (Sargent and Westerberg 1964), whereas Simula-67—the first object-oriented programming language—is described in (Birtwistle, Dahl, Myhrhaug, and Nygaard 1974). The early CSSL language specification is described in (Augustin, Fineberg, Johnson, Linebarger, Sansom, and Strauss 1967) whereas the ACSL system is described in (Mitchell and Gauthier 1986). The Hibliz system for an hierarchical graphical approach to modeling is presented in (Elmqvist 1982).

Software component systems are presented in (Assmann 2002; Szyperski 1997).

The Simulink system for block oriented modeling is described in (MathWorks 2001), whereas the MATLAB language and tool are described in (MathWorks 2002).

The DrModelica electronic notebook with the examples and exercises of this book has been inspired by DrScheme (Felleisen, Findler, Flatt, and Krishnamurthi 1998) and DrJava (Allen, Cartwright, and Stoler 2002), as well as by Mathematica (Wolfram 1997), a related electronic book for teaching mathematics (Davis, Porta, and Uhl 1994), and the MathModelica environment (Fritzson, Engelson, and Gunnarsson 1998; Fritzson, Gunnarsson, and Jirstrand 2002). The first version of DrModelica is described in (Lengquist-Sandelin and Monemar 2003; Lengquist-Sandelin, Monemar, Fritzson, and Bunus 2003).

General Modelica articles and books: (Elmqvist 1997; Fritzson and Engelson 1998; Elmqvist, Mattson, and Otter 1999), a series of 17 articles (in German) of which (Otter 1999) is the first, (Tiller 2001; Fritzson and Bunus 2002; Elmqvist, Otter, Mattsson, and Olsson 2002).

The proceedings from the following conferences, as well as some not listed here, contain a number of Modelica related papers: the Scandinavian Simulation Conference: (Fritzson 1999), and especially the International Modelica Conference: (Fritzson 2000; Otter 2002; Fritzson 2003).

2.24 Exercises

Exercise 2-1:

Question: What is a class?

Creating a Class: Create a class, `Add`, that calculates the sum of two parameters, which are `Integer` numbers with given values.

Exercise 2-2:

Question: What is an instance?

Creating Instances:

```
class Dog
  constant Real legs = 4;
  parameter String name = "Dummy";
end dog;
```

- Create an instance of the class `Dog`.
- Create another instance and give the dog the name "Tim".

Exercise 2-3:

Write a function, `average`, that returns the average of two `Real` values.

Make a function call to `average` with the input 4 and 6.

Exercise 2-4:

Question: What do the terms `partial`, `class`, and `extends` stand for?

Exercise 2-5:

Inheritance: Consider the `Bicycle` class below.

```
record Bicycle
  Boolean has_wheels = true;
  Integer nrOfWheels = 2;
end Bicycle;
```

Define a record, `ChildrensBike`, that inherits from the class `Bicycle` and is meant for kids. Give the variables values.

Exercise 2-6:

Declaration Equations and Normal Equations: Write a class, `Birthyear`, which calculates the year of birth from this year together with a person's age.

Point out the declaration equations and the normal equations.

Modification Equation: Write an instance of the class `Birthyear` above. The class, let's call it `MartinsBirthyear`, shall calculate Martin's year of birth, call the variable `martinsBirthyear`, who is a 29-year-old. Point out the modification equation.

Check your answer, e.g. by writing as below[12]

`martinsBirthday.birthYear`, or `martinsBirthday.birthYear[0]`

[12] In the optional interface to MathModelica supporting Mathematica syntax for commands, the expression `martinsBirthday.birthYear[0]` means the `birthYear` value at time=0, at the beginning of the simulation. It is also in many cases possible to interactively enter an expression such as `martinsBirthday.birthYear` and get back the result without giving the time argument.

Exercise 2-7:

Classes:

```
class Ptest
  parameter Real x;
  parameter Real y;
  Real z;
  Real w;
equation
  x + y = z;
end Ptest;
```

Question: What is wrong with this class? Is there something missing?

Exercise 2-8:

Create a record containing several vectors and matrices:

- A vector containing the two `Boolean` values `true` and `false`.
- A vector with five `Integer` values of your choice.
- A matrix of three rows and four columns containing `String` values of your choice.
- A matrix of one row and five columns containing different `Real` values, also those of your choice.

Exercise 2-9:

Question: Can you really put an algorithm section inside an equation section?

Exercise 2-10:

Writing an Algorithm Section: Create the class, `Average`, which calculates the average between two integers, using an algorithm section.

Make an instance of the class and send in some values.

Simulate and then test the result of the instance class by writing `instanceVariable.classVariable`.

Exercise 2-11: (A harder exercise)

Write a class, `AverageExtended`, that calculates the average of 4 variables (`a, b, c,` and `d`).

Make an instance of the class and send in some values.

Simulate and then test the result of the instance class by writing `instanceVariable.classVariable`.

Exercise 2-12:

If-equation: Write a class `Lights` that sets the variable switch (integer) to one if the lights are on and zero if the lights are off.

When-equation: Write a class `LightSwitch` that is initially switched off and switched on at time 5.

Tip: `sample(start, interval)` returns true and triggers time events at time instants and `rem(x, y)` returns the integer remainder of `x/y`, such that `div(x,y) * y + rem(x, y) = x`.

Exercise 2-13:

Question: What is a Package?

Creating a Package: Create a package that contains a division function (that divides two `Real` numbers) and a constant `k = 5`.

Create a class, containing a variable `x`. The variable gets its value from the division function inside the package, which divides 10 by 5.

Part II

The Modelica Language

Chapter 3

Classes, Types, and Declarations

The fundamental unit of modeling in Modelica is the class. Classes provide the structure for objects, also known as instances, and serve as templates for creating objects from class definitions. Classes can contain equations which provide the basis for the executable code that is used for computation in Modelica. Conventional algorithmic code can also be part of classes. Interaction between objects of well-structured classes in Modelica is usually done through so-called connectors, which can be seen as "access ports" to objects. All data objects in Modelica are instantiated from classes, including the basic data types—`Real`, `Integer`, `String`, `Boolean`—and enumeration types, which are built-in classes or class schemata.

A class in Modelica is essentially equivalent to a type. Declarations are the syntactic constructs needed to introduce classes and objects.

This chapter is organized into two parts. The first part comprising sections 3.1 to 3.6 gives a tutorial introduction to the Modelica class concept using the Moon landing example as well as the contract idea. The rest of the chapter is presented in a somewhat more formal style, giving a complete description of most aspects of Modelica classes, types, declarations, and lookup, except inheritance covered in Chapter 4 and component aspects covered in Chapter 5.

3.1 Contract Between Class Designer and User

Object-oriented modeling languages try to separate the notion of *what* an object is from *how* its behavior is implemented and specified in detail. The "what" of an object in Modelica is usually described through documentation including graphics and icons, together with possible public connectors, variables, and other public elements, and their associated semantics. For example, the "what" of an object of class `Resistor` is the documentation that it models a "realistic" ideal resistor coupled with the fact that its interaction with the outside world is through two connectors `n`, `p` of type `Pin`, and their semantics. This combination—documentation, connectors and other public elements, and semantics—is often described as a *contract* between the designer of the class and the modeler who uses it, since the "what" part of the contract specifies to the modeler what a class represents, whereas the "how" provided by the class designer implements the required properties and behavior.

An incorrect, but common, assumption is that the connectors and other public elements of a class (its "signature") specify its entire contract. This is not correct since the intended semantics of the class is also part of the contract even though it might only be publicly described in the documentation, and internally modeled through equations and algorithms. Two classes e.g., a `Resistor` and a temperaturedependent `Resistor`, may have the same "signature" in terms of connectors but still not be equivalent since they have different semantics. The contract of a class includes both the signature and the appropriate part of its semantics together.

The “how” of an object is defined by its class. The implementation of the behavior of the class is defined in terms of equations and possibly algorithmic code. Each object is an instance of a class. Many objects are composite objects i.e., consist of instances of other classes.

3.2 A Class Example

The basic properties of a class are given by:

- Data contained in variables declared in the class.
- Behavior specified by equations together with possible algorithms.

Here is a simple class called `CelestialBody` that can be used to store data related to celestial bodies such as the earth, the moon, asteroids, planets, comets, and stars:

```
class CelestialBody
  constant  Real   g = 6.672e-11;
  parameter Real   radius;
  parameter String name;
  Real             mass;
end CelestialBody;
```

The declaration of a class starts with a keyword such as class, model, etc., followed by the name of the class. A class declaration creates a *type name* in Modelica, which makes it possible to declare variables of that type, also known as objects or instances of that class, simply by prefixing the type name to a variable name:

```
CelestialBody moon;
```

This declaration states that `moon` is a variable containing an object of type `CelestialBody`. The declaration actually creates the object i.e., allocates memory for the object. This is in contrast to a language like Java, where an object declaration just creates a reference to an object.

This first version of `CelestialBody` is not very well designed. This is intentional, since we will demonstrate the value of certain language features for improving the class in this and the following two chapters.

3.3 Variables

The variables belonging to a class are sometimes called record fields or attributes; the `CelestialBody` variables `radius`, `name`, and `mass` are examples. Every object of type `CelestialBody` has its own instances of these variables. Since each separate object contains a different instance of the variables this means that each object has its own unique state. Changing the `mass` variable in one `CelestialBody` object does not affect the `mass` variables in other `CelestialBody` objects.

Certain programming languages e.g., Java and C++, allow so-called static variables, also called class variables. Such variables are shared by all instances of a class. However, this kind of variable is not available in Modelica.

A declaration of an instance of a class e.g., `moon` being an instance of `CelestialBody`, allocates memory for the object and initializes its variables to appropriate values. Three of the variables in the class `CelestialBody` have special status: the gravitational *constant* `g` is a `constant` that never changes, and can be substituted by its value. The *simulation parameters* `radius` and `name` are examples of a special kind of “constant” denoted by the keyword `parameter` in Modelica. Such simulation parameter “constants” are assigned their values only at the start of the simulation and keep their values constant during simulation.

In Modelica variables store results of computations performed when solving the equations of a class together with equations from other classes. During solution of timedependent problems, the variables store results of the solution process at the current time instant.

As the reader may have noted, we use the terms *object* and *instance* interchangeably with the same meaning; we also use the terms *record field*, *attribute*, and *variable* interchangeably. Sometimes the *term* variable is used interchangeably with *instance* or *object*, since a variable in Modelica always contains an instance of some class.

3.3.1 Duplicate Variable Names

Duplicate variable names are not allowed in class declarations. The name of a declared element e.g., a variable or local class, must be different from the names of all other declared elements in the class. For example, the following class is illegal:

```
class IllegalDuplicate
  Real    duplicate;
  Integer duplicate;    // Error! Illegal duplicate variable name
end IllegalDuplicate;
```

3.3.2 Identical Variable Names and Type Names

The name of a variable is not allowed to be identical to its type specifier. Consider the following erroneous class:

```
class IllegalTypeAsVariable
  Voltage Voltage;   // Error! Variable name must be different from type
  Voltage voltage;   // Ok! Voltage and voltage are different names
end IllegalTypeAsVariable;
```

The first variable declaration is illegal since the variable name is identical to the type specifier of the declaration. The reason this is a problem is that the `Voltage` type lookup from the second declaration would be hidden by a variable with the same name. The second variable declaration is legal since the lowercase variable name `voltage` is different from its uppercase type name `Voltage`.

3.3.3 Initialization of Variables

The default suggested (the solver may choose otherwise, if not `fixed`) initial variable values are the following, if no explicit start values are specified (see Section 2.3.2, page 27, and Section 8.4 below on page 250):

- The value zero as the default initial value for numeric variables.
- The empty string `""` for String variables.
- The value `false` for `Boolean` variables.
- The lowest enumeration value in an enumeration type for enumeration variables.

However, *local variables* to functions have *unspecified* initial values if no defaults are explicitly given. Initial values can be explicitly specified by setting the `start` attributes of instance variables equal to some value, or providing initializing assignments when the instances are local variables or formal parameters in functions. For example, explicit start values are specified in the class `Rocket` shown in the next section for the variables `mass`, `altitude`, and `velocity`.

3.4 Behavior as Equations

Equations are the primary means of specifying the behavior of a class in Modelica, even though algorithms and functions also are available. The way in which the equations interact with equations from other classes determines the solution process i.e., program execution, where successive values of variables are computed over time. This is exactly what happens during dynamic system simulation. During solution of timedependent problems, the variables store results of the solution process at the current time instant.

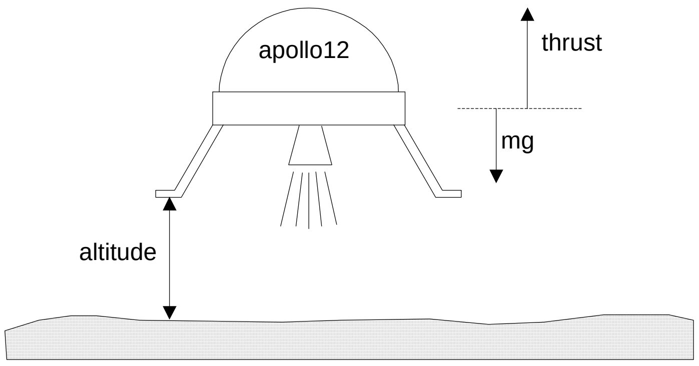


Figure 3-1. Apollo12 rocket for landing on the moon.

The class `Rocket` embodies the equations of vertical motion for a rocket (e.g. as depicted in Figure 3-1) which is influenced by an external gravitational force field `gravity`, and the force `thrust` from the rocket motor, acting in the opposite direction to the gravitational force, as in the expression for acceleration below:

$$acceleration = \frac{thrust - mass \cdot gravity}{mass}$$

The following three equations are first-order differential equations stating well-known laws of motion between altitude, vertical velocity, and acceleration:

$$mass' = -massLossRate \cdot abs(thrust)$$
$$altitude' = velocity$$
$$velocity' = acceleration$$

All these equations appear in the class `Rocket` below, where the mathematical notation (′) for derivative has been replaced by the pseudofunction `der`() in Modelica. The derivative of the rocket `mass` is negative since the rocket fuel mass is proportional to the amount of `thrust` from the rocket motor.

```
class Rocket "rocket class"
  parameter String name;
  Real mass(start=1038.358);
  Real altitude(start= 59404);
  Real velocity(start= -2003);
  Real acceleration;
  Real thrust;    // Thrust force on the rocket
  Real gravity;   // Gravity forcefield
  parameter Real massLossRate=0.000277;
equation
  (thrust-mass*gravity)/mass = acceleration;
  der(mass)   = -massLossRate * abs(thrust);
  der(altitude) = velocity;
  der(velocity) = acceleration;
end Rocket;
```

The following equation, specifying the strength of the gravitational force field, is placed in the class `MoonLanding` in the next section since it depends on both the mass of the rocket and the mass of the moon:

$$gravity = \frac{g_{moon} \cdot mass_{moon}}{\left(altitude_{apollo} + radius_{moon}\right)^2}$$

The amount of thrust to be applied by the rocket motor is specific to a particular class of landings, and therefore also belongs to the class `MoonLanding`:

$$\begin{aligned} thrust = if\ &(time < thrustDecreaseTime)\ then \\ &\quad force1 \\ else\ if\ &(time < thrustEndTime)\ then \\ &\quad force2 \\ else\ &0 \end{aligned}$$

3.5 Access Control

Members of a Modelica class can have two levels of visibility: `public` or `protected`. The default is `public` if nothing else is specified e.g., regarding the variables `force1` and `force2` in the class `MoonLanding` below. The `public` declaration of `force1`, `force2`, `apollo`, and `moon` means that any code with access to a `MoonLanding` instance can read or update those values.

The other possible level of visibility, specified by the keyword `protected`—e.g., for the variables `thrustEndTime` and `thrustDecreaseTime`, means that only code *inside* the class as well as code in classes that inherit this class are allowed access. Code inside a class includes code from local classes, see Section 3.7.2 on page 80. However, only code inside the class is allowed access to the *same* instance of a protected variable—classes that extend the class will naturally access another instance of a protected variable since declarations are "copied" at inheritance. This is different from corresponding access rules for Java.

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.

The variables `thrust`, `gravity`, and `altitude` belong to the `apollo` instance of the `Rocket` class and are therefore prefixed by `apollo` in references such as `apollo.thrust`. The gravitational constant `g`, the `mass`, and the `radius` belong to the particular celestial body called `moon` on which surface the `apollo` rocket is landing.

```
class MoonLanding
  parameter Real force1 = 36350;
  parameter Real force2 = 1308;
  protected
  parameter Real thrustEndTime = 210;
  parameter Real thrustDecreaseTime = 43.2;
public
  Rocket          apollo(name="apollo12");
  CelestialBody   moon(name="moon",mass=7.382e22,radius=1.738e6);
equation
  apollo.thrust = if (time<thrustDecreaseTime) then force1
                  else if (time<thrustEndTime) then force2
                  else 0;
  apollo.gravity = moon.g * moon.mass / (apollo.altitude + moon.radius)^2;
end MoonLanding;
```

3.6 Simulating the Moon Landing Example

We simulate the `MoonLanding` model during the time interval `{0,230}` by the following command, using the MathModelica simulation environment:

```
simulate(MoonLanding, stopTime=230)
```

Since the solution for the altitude of the Apollo rocket is a function of time, it can be plotted in a diagram (Figure 3-2). It starts at an altitude of 59404 (not shown in the diagram) at time zero, gradually reducing it until touchdown at the lunar surface when the altitude is zero. Note that a MathModelica `plot` or `PlotSimulation` command refers to the results of the most *recent* simulation. If there is need to plot results of some previous simulation e.g., from a Mars landing, the value returned by the `simulate` command can be stored in a variable that can be referred to (in the Mathematica command interface to MathModelica by the optional `SimulationResult` in the `PlotSimulation` command).

```
plot(apollo.altitude, xrange={0,208})
```

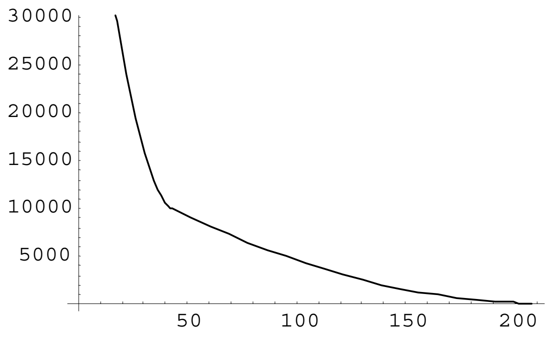


Figure 3-2. Altitude of the Apollo rocket over the lunar surface.

The thrust force from the rocket is initially high but is reduced to a low level after 43.2 seconds i.e., the value of the simulation parameter `thrustDecreaseTime`, as shown in Figure 3-3.

```
plot(apollo.thrust, xrange={0,208})
```

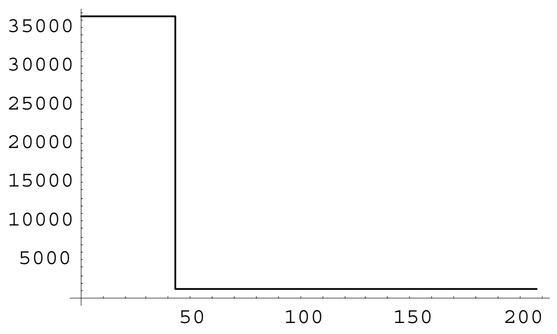


Figure 3-3. Thrust from the rocket motor, with an initial high thrust `f1` followed by a lower thrust `f2`.

The mass of the rocket decreases from initially 1038.358 to around 540 as the fuel is consumed (Figure 3-4).

```
plot(apollo.mass, xrange={0,208})
```

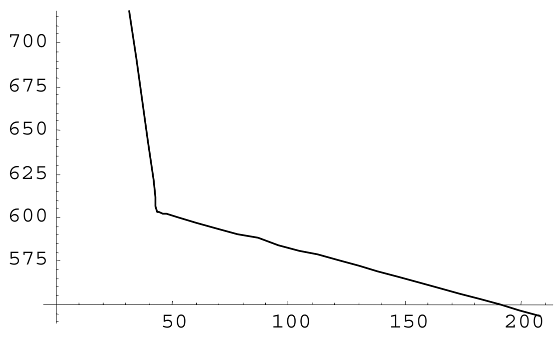


Figure 3-4. Rocket mass decreases when the fuel is consumed.

The gravity field increases when the rocket gets closer to the lunar surface, as depicted in Figure 3-5, where the gravity has increased to 1.63 after 200 seconds.

```
plot(apollo.gravity, xrange={0,208})
```

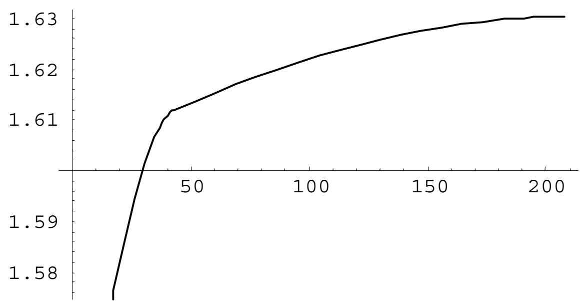


Figure 3-5. Gradually increasing gravity when the rocket approaches the lunar surface.

The rocket initially has a high negative velocity when approaching the lunar surface. This is reduced to zero at touchdown, giving a smooth landing, as shown in Figure 3-6.

```
plot(apollo.velocity, xrange={0,208})
```

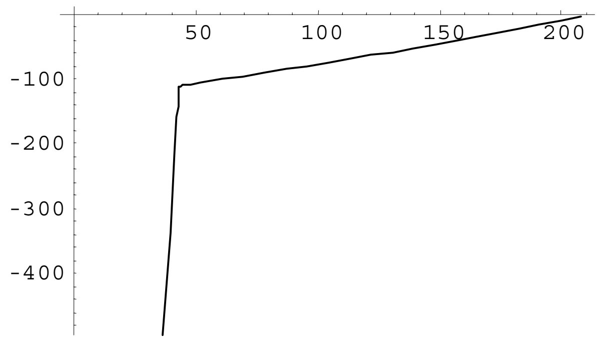


Figure 3-6. Vertical velocity relative to the lunar surface.

When experimenting with the `MoonLanding` model, the reader might notice that the model is nonphysical regarding at least one important aspect of the landing. After touchdown when the speed has been reduced to zero, if the simulation is allowed to continue the speed will increase again and the lander will accelerate toward the center of the moon. This is because we have left out the ground contact force from the lunar surface acting on the lander after it has landed that will prevent this from happening. It is left as an exercise to introduce such a ground force into the `MoonLanding` model.

3.6.1 Object Creation Revisited

As already pointed out, objects are created implicitly in Modelica just by declaring instances of classes. This is done simply by prefixing the type name to the variable name e.g., the variable `apollo` of type `Rocket` below. This is in contrast to object-oriented languages like Java or C++, where object creation on the heap is specified using the `new` keyword.

```
Rocket apollo;
```

There is one remaining problem, however. In what context i.e., within which class, should the `apollo` object be instantiated, and how do we ensure that it actually becomes instantiated? It could be that the `MoonLanding` class is just a library class, which is not instantiated if not part of some existing instance.

As explained previously in Section 2.3.1 on page 26, this problem is solved by regarding the class specified in the simulation command as a kind of “main” class that is always instantiated, implying that its variables are instantiated, and that the variables of those variables are instantiated, etc. Therefore, to instantiate `Rocket` as an object called `apollo`, it is declared as a variable in the simulation model to be simulated, which in this example happens to be `MoonLanding`.

3.7 Short Classes and Nested Classes

Modelica has a very concise way of declaring classes, called *short class definition*. It is also possible to declare classes locally within another class, socalled *nested classes*.

3.7.1 Short Class Definitions

There is a short syntax for defining classes in terms of other classes, commonly used to introduce more informative type names for existing classes, which, for example, is similar to the typedef construct in the C language. For example:

```
class Voltage = Real;
```

It is possible to use the alternative keyword `type` instead of class:

```
type Voltage = Real;
```

The short class definition is the only way of introducing a “class” name for an array type, e.g.:

```
type Matrix10 = Real[10,10];
```

3.7.2 Local Class Definitions—Nested Classes

In Modelica it is possible to define local classes nested inside a class, to an arbitrary level of nesting if desired. For example, the classes `Lpin` and `Voltage` below are locally defined:

```
class C1
  class Lpin
    Real p;
  end Lpin;

  class Voltage = Real(unit="kV");

  Voltage v1, v2;
  Lpin    pn;
end C1;
```

All declarations of a class, even `protected` ones, are accessible from within a local nested class, except when that local class or a class enclosing that class is encapsulated. For more details, see the rules for scoping and lookup described in Section 3.13 on page 96.

3.8 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. A `block` is a `class` with fixed causality i.e., each part of its interface must have causality equal to `input` or `output`. A `connector` class is used to declare the structure of connection points 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.

A class that inherits a restricted class needs to be of the same restricted class kind, or have the same or more restrictions as the parent class, which is described in more detail in Chapter 4, page 116.

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 useful since the modeler does not have to learn several completely different concepts, but only one: the class concept. Many properties of a class, such as the syntax and semantics of definition, instantiation, inheritance, and generic properties, are identical to 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. One can think of those as restricted and enhanced “classes.”

Below we briefly summarize the typical usage, restrictions, and enhancements for each kind of restricted class compared to the generic Modelica class construct, as well as giving a few simple examples.

3.8.1 Restricted Class: model

The `model` restricted class is the most commonly used kind of class for modeling purposes. Its semantics are almost identical to the general class concept in Modelica. The only restriction is that a `model` may *not be used in connections*. The previous example classes `CelestialBody`, `Rocket`, and `MoonLanding` could very well have been defined as models. A trivial example:

```
model Resistor "Electrical resistor"
  Pin p, n  "positive and negative pins";
  Voltage v;
  Current i;
  parameter Real R(unit="Ohm")          "Resistance";
equation
  v = i*R;
end Resistor;
```

3.8.2 Restricted Class: record

A `record` is a class for specifying data without behavior. *No equations* are allowed in the definition of a record or in any of its elements e.g., in local classes. A record *may not be used in connections*. Those restrictions also apply to classes that inherit a record, since classes with less restrictions are not allowed to inherit a record class. An example:

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

3.8.3 Restricted Class: type

The `type` restricted class is typically used to introduce new type names using short class definitions, even though short class definitions can be used to introduce new names also for other kinds of classes. A `type` restricted class may be only an *extension* to one of the *predefined types*, a *record* class, or an *array* of some type. Note that all kinds of classes define types in Modelica, not only those using the `type` keyword. The general concepts of Modelica types and type checking are described in Section 3.14 on page 100. Example:

```
type Matrix = Real[3,3];
```

3.8.4 Restricted Class: connector

Connector classes are typically used as templates for connectors i.e., a kind of “port” for communicating data between objects. Therefore *no equations* are allowed in the definition of a `connector` class or within any of its elements. We examine connector classes, connectors, and connections in Chapter 5. The following connector class, called `Flange`, is from the standard Modelica mechanical library for translational motion:

```
connector Flange
  Position    s;
  flow Force  f;
end Flange;
```

3.8.5 Restricted Class: block

A `block` is a class for which the causality i.e., data-flow direction being either input or output, is known for each of its variables. Thus, blocks are typically used for signalrelated modeling since in that case the data-flow direction is always known. The main restriction of a block is that all *variables* declared in a block must have one of the *prefixes* `input` or `output`. A block also *may not be used in connections*. A trivial example:

```
block Multiply
  input  Real x;
  input  Real y(start=0);
  output Real result;
equation
  result = x*y;
end Multiply;
```

3.8.6 function

The `function` concept in Modelica corresponds to mathematical functions without external side effects, and can be regarded as a restricted class with some enhancements. Its syntax and semantics are close to that of the `block` restricted class. Function results are returned by those formal parameters prefixed by the keyword `output`. A `function` has the following restrictions compared to a `class`:

- Each formal parameter of the function must be *prefixed* by either `input` or `output` and be public and nonconstant i.e., the same as for `block`.
- A function may not be used in connections, may have no equations, and can have at most one algorithm.
- For a function to be called, it must have either an algorithm or an external function interface as its body.
- A function cannot contain calls to the Modelica built-in operators `der`, `initial`, `terminal`, `sample`, `pre`, `edge`, `change`, `reinit`, `delay`, and `cardinality`, and is not allowed to contain when-statements.
- A subtype of a function type needs to be equivalent to the function type itself.
- The size of each array result or array local variable of a function must be either given by the formal parameters, or given by constant or parameter expressions.

A `function` has the following additional properties: it may be called using the conventional *positional calling syntax* for passing arguments, it may be a *recursive*, and a formal parameter or local variable may be initialized through an *assignment* of a default value in its declaration. A function is dynamically instantiated when it is called. The previous `Multiply` example, formulated as a function, where we have changed the name to `multiply`, following the convention that a function name should start with a lowercase letter:

```
function multiply
  input  Real x;
  input  Real y := 0;
  output Real result;
algorithm
  result := x*y;
end multiply;
```

Additional information regarding Modelica functions is available in Chapter 8, page 298.

3.8.7 package

A `package` is a restricted and enhanced class that is primarily used to manage name spaces and organize Modelica code. A package has the restrictions of only *containing* declarations of *classes* including all kinds of restricted classes, and *constants* i.e., no variable declarations. A package has the enhancement of being allowed to *import* from. The prefix `encapsulated` makes a package an independent unit of code where dependencies to definitions in other packages are explicitly visible through `import` clauses. A trivial example package:

```
encapsulated package SmallPack
  import OtherPackage;
  constant Real mypi = 3.14159;

  connector Pin
     ...
  end Pin;

  function multiply
```

```
      ...
    end multiply;
  end SmallPack;
```

More information on packages is available in Chapter 10.

3.9 Predefined Types/Classes

The basic predefined built-in types of Modelica are `Real`, `Integer`, `Boolean`, `String`, and the basic enumeration type. These types have some of the properties of a class e.g., can be inherited, modified, etc., but on the other hand have a number of restrictions and somewhat different properties compared to normal classes. At type checking variables of these types are regarded as scalar types. Only the *value* attribute can be changed at run-time, and is accessed through the variable name itself, and not through dot notation e.g., use `x` and not `x.value` to access the value of the variable `x`. The primitive types of the machine representations of the *values* of these predefined types have the following properties:

`Real`	IEC 60559:1989 (ANSI/IEEE 754-1985) double format, at least 64-bit precision.
`Integer`	typically two's-complement 32-bit integer.
`Boolean`	`true` or `false`.
`String`	string of 8-bit characters.
Enumeration types	Distinct from `Integer` but values are internally represented as a positive integers.

Note that for argument passing of values when calling external functions in C from Modelica, `Real` corresponds to `double` and `Integer` corresponds to `int`. Below, we use '`RealType`', '`BooleanType`', '`IntegerType`', '`StringType`', and '`EnumType`' as mnemonics corresponding to the above-mentioned primitive types. These mnemonics are enclosed in single quotes to emphasize that they are not available in the Modelica language.

Apart from being built-in types containing machine representations of values, the predefined Modelica types can also be inherited and modified similarly to ordinary Modelica classes, but with the restrictions that variables and equations *cannot be added* in the derived class (i.e., the class that inherits/extends some other class) and that attributes cannot be redeclared (see Chapter 4 regarding redeclaration). Apart from value attribute, the predefined types also have a number of additional attributes that are parameters i.e., cannot be modified at run-time during simulation. These parameter attributes are accessed through ordinary dot notation e.g., `x.unit` for the attribute unit of the variable `x`. The only way to define a subtype of a predefined Modelica type is through inheritance, usually expressed as modifications.

For example, a `Real` variable has a set of attributes such as unit of measure, initial value, minimum and maximum value. Some of these attributes can be modified when declaring a new type, for example, through a short type definition:

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

The same definition can also be expressed using the standard inheritance syntax (see Chapter 4 for more details):

```
  type Voltage
    extends Real(unit= "V", min=-220.0, max=220.0);
  end Voltage;
```

The attributes i.e., the class attributes, of the predefined variable types are described below with Modelica syntax as part of pseudo-type class declarations. These are "pseudodeclarations" since they cannot be declared and processed by standard Modelica. Also, as already stated, redeclaration of any of the predefined attributes is not allowed. If the values of these attributes are not explicitly specified, the defaults given below are used.

3.9.1 Real Type

The predefined `Real` type is described by the following pseudoclass declaration:

```
type Real "Pseudo class declaration of predefined Real type"
  'RealType'   'value';       // Accessed without dot-notation
  parameter 'StringType'  quantity    = "";
  parameter 'StringType'  unit        = ""  "Unit used in equations";
  parameter 'StringType'  displayUnit = ""  "Default display unit";
  parameter 'RealType'    min=-Inf, max=+Inf; //Inf denotes a large value
  parameter 'RealType'    start = 0;      // Initial value
  parameter 'BooleanType' fixed = true,  // default for parameter/constant
                                = false; // default for other variables
  parameter 'RealType'    nominal;       // Nominal value equation
  parameter StateSelect   stateSelect = StateSelect.default;
equation
  assert('value' >= min and 'value' <= max,"Variable value out of limit");
  assert(nominal >= min and nominal <= max, "Nominal value out of limit");
end Real;
```

The attributes of the `Real` predefined type have the following meaning:

- *'value'*—not a class attribute in the usual sense but rather the scalar value of a variable declared as having the type `Real`. The value is accessible directly via the variable name and is not accessible using dot notation, which is indicated by enclosing this attribute within single quotes. Thus, use `x` and not `x.value` to access the value of the variable `x`.
- *quantity*—a string describing what physical quantity the type represents. Example quantities could be "force," "weight," "acceleration," etc.
- *unit*—a string describing the physical unit of the type. Examples are `"kg"`, `"m"`, `"m/s"`, etc. See also Section 11.7.1, page 375.
- *displayUnit*—gives the default unit used by a Modelica tool for interactive input/output. For example, the `displayUnit` for angle might be `"deg"`, but the `unit` might be `"rad"`. Values interactively input or output as `"deg"` are then automatically converted to/from `"rad"`. See also 11.7.1, page 375.
- *min*—the minimum scalar value representable by the type.
- *max*—the maximum scalar value representable by the type.
- *start*—the initial value of a variable declared as being an instance of the type. The start value is assigned to the variable before starting time-dependent simulation. See also Section 8.4 belowon page 250.
- *fixed*—determines whether the `start` value of a variable having this type is fixed or can be adjusted by the simulator before actually starting the simulation. The default for the attribute `fixed` is `true` for constants and parameters but `false` for other variables, which in practice means (unless `fixed` is explicitly set to `true`) that the specified start value is treated as an initial guess for the variable that can be changed by the simulator in order to find a consistent set of start values e.g., before proceeding with dynamic simulation forward in time. See also Section 3.11.11, page 94.
- *nominal*—a nominal value of the type i.e., a "typical" value that can be used, e.g., by a simulation tool to determine appropriate tolerances or for scaling purposes by giving a hint of the order of magnitude for typical values. The user need not set this value even though a default value is not specified. The reason is that in cases such as `a=b`, where `b` has a nominal value but not `a`, the nominal value can be propagated to the other variable.

- *stateSelect*—this attribute is used for manual control of which variables should be selected as state variables for numerical solution of equation systems (see Section 18.2.6.1, page 686, for more information). The default value for this attribute is the enumeration value `StateSelect.default`, which gives automatic state variable selection—this usually works well in almost all applications and the user should therefore normally not need to set this attribute.

3.9.2 Integer, Boolean, and String Types

Analogously to the `Real` predefined type, pseudoclass declarations for the types `Integer`, `Boolean`, and `String` are given below:

```
type Integer "Pseudo class declaration of predefined Integer type"
  'IntegerType' 'value';      // Accessed without dot-notation
  parameter 'IntegerType' min=-Inf, max=+Inf;
  parameter 'IntegerType' start = 0;      // Initial value
  parameter 'BooleanType' fixed = true,  // default for parameter/constant;
                               = false; // default for other variables
equation
  assert('value'= min and 'value'<= max, "Variable value out of limit");
end Integer;

type Boolean "Pseudo class declaration of predefined Boolean type"
  'BooleanType' 'value';       // Accessed without dot-notation
  parameter 'BooleanType' start = false; // Initial value
  parameter 'BooleanType' fixed = true,  // default for parameter/constant
                               = false; // default for other variables
end Boolean;

type String "Pseudo class declaration of predefined String type"
          'StringType' 'value';       // Accessed without dot-notation
  parameter 'StringType' start = "";   // Initial value
end String;
```

3.9.3 Enumeration Types

Conceptually an enumeration type is a type whose values can be enumerated using a finite enumeration. In theory, finite ranges of integers are also enumeration types.

However, in Modelica we reserve the concept of *enumeration type* for types defined using a special enumeration type construct, in which each possible enumeration value of the type is explicitly named. For example, an enumeration type `ShirtSizes` can be defined as follows:

```
type ShirtSizes = enumeration(small, medium, large, xlarge);
```

Enumeration types are regarded as *simple predefined types* with many properties in common with integers apart from the fact that their values, so-called enumeration literals, are explicitly named.

An optional comment string can be specified with each enumeration literal in the enumeration type declaration, e.g., as follows:

```
type ShirtSizes = enumeration(small "1st", medium "2nd", large "3rd",
                              xlarge "4th");
```

Here the enumeration literals are: `ShirtSizes.small`, `ShirtSizes.medium`, `ShirtSizes.large`, and `ShirtSizes.xlarge`.

To show a simple example of using this type we declare a variable `my_shirtsize` of enumeration type `ShirtSizes` and set its start value to the constant enumeration literal value `ShirtSizes.medium`:

```
ShirtSizes my_shirtsize(start = ShirtSizes.medium);
```

Values of enumeration types can be used in many ways analogous to integer values e.g., for indexing, comparison, etc., but excluding arithmetic operations. Iteration over ranges of enumeration values is especially useful (Section 9.2.5, page 288). Arrays indexed by enumeration values can currently only be used in two ways: array element indexing (Section 7.4.1.1, page 217), or in array declaration equations or assignments (Section 7.6.1, page 222). Enumeration values can be stored as array elements. It is possible to use an enumeration type name to represent the possible index elements of array dimensions in declarations (see Section 7.6.1, page 222 and this example):

```
Real w[ShirtSizes]  "Weight of shirts depending on size";
```

There exists a special enumeration type which is a supertype of all possible enumeration types:

```
type SuperEnumeration = enumeration(:);
```

The only practical use of the special enumeration type `enumeration(:)` is as a constraining type of replaceable enumeration types (see Section 4.3.5, page 129), and in enumeration subtyping rules. In general we declare an enumeration type *E* as follows:

```
type E = enumeration(e1, e2, ..., en);
```

where the enumeration literals *e1*, *e2*, ... *en* must have different names within the enumeration type. The names of the enumeration literals are defined inside the scope of the enumeration type *E*. Each enumeration literal in the list has type *E*.

The enumeration type *E* can be viewed as conceptually defined according to the following predefined type schema:

```
type E    "Pseudo class declaration of any enumeration type E"
  'EnumType' 'value';                      // Accessed without dot-notation
  parameter 'StringType'  quantity  = "";
  parameter 'EnumType'    min=e1, max=en;
  parameter 'EnumType'    start = e1;    // Initial value
  parameter 'BooleanType' fixed = true,  // default for parameter/constant;
                                = false; // default for other variables
  constant  'EnumType'    e1 =...;
  ...
  constant  'EnumType'    en =...;
equation
  assert('value' >= min and 'value' <= max, "Variable value out of limit");
end E;
```

However, even though the above enumeration type schema is similar to an ordinary class declaration, enumeration types are *subrange types* that do not follow the usual subtyping rules for classes (see Section 3.14.1, page 102, and Section 3.14.11, page 108, for more details).

One can obtain the ordinal integer value of an enumeration value by using `Integer` as a conversion function. The call `Integer(E.enumvalue)` returns the ordinal number of the enumeration value `E.enumvalue`, where `Integer(E.e1)=1` and `Integer(E.en)=`*n*, for an enumeration type *E* = `enumeration(e1, e2, ..., en)`. Referring to our previous example, `Integer(ShirtSizes.Medium) = 2`.

There is also a conversion function `String` that returns the string representation of an enumeration value. For example, `String(ShirtSizes.Large) = "Large"`.

3.9.4 Other Predefined Types

The predefined `StateSelect` enumeration type is the type of the `stateSelect` attribute of the `Real` type. It is used to control state selection, and is defined and explained in Section 18.2.6.1, page 686.

The predefined partial class `ExternalObject` is used to define opaque references to data structures created outside Modelica (see Section 9.4.6, page 320).

A number of "predefined" record types and enumeration types for graphical annotations are described in Chapter 11. However these types are not predefined in the usual sense since they cannot be referenced in ordinary Modelica code, only within annotations.

3.10 Structure of Variable Declarations

A Modelica variable i.e., a class instance, is always a member of some other class instance or function instance. The declaration of a variable states the type, access, variability, data flow, and other properties of the variable. A declaration can be divided into three parts: zero or more *prefixes*, followed by a *type*, followed by a list of *variable specifiers*. A variable specifier can be a single *identifier*, or an identifier optionally followed by a *modifier* and/or an array dimension descriptor, optionally followed by an *initializer* or *declaration equation* right-hand side. Thus, the structure of a *declaration* is:

<prefixes> <type-specifier> <variable specifier list>

A *variable specifier* has the following structure, where all elements apart from the identifier are optional:

<identifier> <array dimension descriptor> <modifiers> <declaration equation/initializer>

The type specifier part of a declaration describes the kind of values that can be represented by the declared variable(s). It consists of a type name optionally followed by an array dimension descriptor.

There is no difference between variables declared in a single declaration or in multiple declarations. For example, regard the following single declaration of two matrix variables:

```
Real[2,2]  A, B;
```

That declaration has the same meaning as the following two declarations together:

```
Real[2,2]  A;
Real[2,2]  B;
```

The array dimension descriptors may instead be placed after the variable name in the variable specifier, giving the two declarations below, with the same meaning as in the previous example:

```
Real  A[2,2];
Real  B[2,2];
```

The following declaration is different, meaning that the variable `a` is a scalar and `B` is a matrix as above:

```
Real  a, B[2,2];
```

3.11 Declaration Prefixes

Both class and variable declarations can contain various qualifier prefixes. These prefixes associate certain properties with the declaration and are placed at the beginning of the declaration. The following sections give an overview of the Modelica declaration prefixes.

3.11.1 Access Control by Prefixes: public and protected

As was already described in the section on access control on page 77, an optional *access prefix* may be specified at the beginning of a declaration. The access prefixes available in Modelica are `public` and `protected`, giving two possible levels of visibility for Modelica variables. This is illustrated by the class `AccessDemo` below.

Note that an access prefix is implicitly *applied* to all *subsequent declarations* without access prefix following the current declaration until the next declaration with an access prefix in the same scope. This makes it natural to view an access prefix as a *heading* for a *section* of declarations. The initial default is `public` if nothing yet has been specified within the current scope. Thus, the variables `a`, `x`, `y`, `z`, `u3` are public, whereas `w`, `u`, and `u2` are `protected`.

```
class AccessDemo "Illustration of access prefixes"
  Real a;
public
  Real x;
  Real y, z;
protected
  Real w, u;
  Real u2;
public
  Real u3;
end AccessDemo;
```

The `public` declaration of variables `a`, `x`, `y`, `z`, `u3` means that any code with access to an `AccessDemo` instance can read or update those variables. The other possible level of visibility, specified by the keyword `protected`, means that only code inside the class `AccessDemo` as well as code in classes that inherit this class are allowed access to the variables `w`, `u`, and `u2`. In fact, only code inside the class `AccessDemo` has access to the same instance of the protected variables `w`, `u`, and `u2`.

Elements are inherited with their own protection, i.e., `public` and `protected` elements keep being `public` and `protected`, respectively, after being inherited into another class. However, the access prefixes `public` and `protected` from a parent class are not inherited as elements themselves, and can therefore never influence the protection status of the declarations in a child class.

3.11.2 Variability Prefixes: constant, parameter and discrete

The prefixes `constant`, `parameter`, `discrete` of a variable declaration are called *variability prefixes* and define in which situations the value of variable may change and when it is kept constant. The term *variability* here should be interpreted as *time variability*. The variability prefixes can be used to define the following four classes of variables (see Figure 3-7):

- Named *constants* i.e., variables with the `constant` prefix, which *never change value* once they have been defined. This can, for example, be used to define mathematical constants. The derivative of a constant is zero. A constant definition is automatically `final` i.e., behaves as if `final` prefix (see Section 3.11.5, page 91) is present in the definition. Therefore a, constant definition can never be redefined, not even an inherited copy of a constant definition.
- Named constant *parameters* i.e., variables with the `parameter` prefix, which *remain constant* during time-dependent simulation and have time derivative equal to zero. These “constants” may be assigned values e.g., during the initialization phase or interactively by the user via a simulation tool immediately before time-dependent simulation starts). Interactive setting of simulation parameters is usually done very conveniently via the graphical user interface of the simulation tool. The built-in `der`(…) derivative operator need not be applied to such variables since the result is always zero.
- *Discrete-time* variables with the optional associated prefix `discrete` are *piecewise constant* signals that change their values only at event instants during simulation (see Section 13.2.2 in Chapter 13), and have time derivative equal to zero. However, it is illegal to apply the `der`() operator to a discrete-time variable. `Boolean`, `Integer`, `String`, and `enumeration` variables are discrete-time by default and cannot be continuous-time. A `Real` variable with the prefix

`discrete` is a discrete-time variable. However, note that a `Real` variable *without* the `discrete` prefix but assigned a value in a when-statement or when-equation is also a discrete-time variable.

- *Continuous-time* variables evolve their values continuously during simulation. This is default for `Real` variables if no variability prefix is specified and they are not assigned in a when-equation or when-statement. This is the most common kind of variable in continuous models.

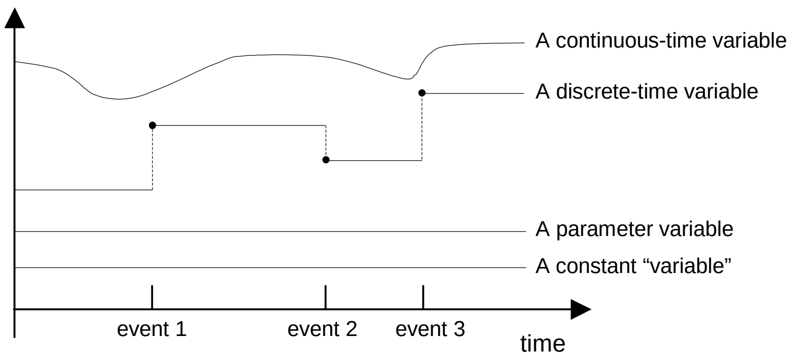


Figure 3-7. Illustrating variables with different variability.

Named constants can be defined in Modelica by using the prefixes `constant` or `parameter` in declarations and providing declaration equations as part of the declarations. For example:

```
constant  Real    PI = 4*arctan(1);
constant  String  redcolor = "red";
constant  Integer one = 1;
parameter Real    mass = 22.5;
constant  Real    tobespecified; // to be given value through a modification
```

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

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

Parameter "constants" may also be used as *structural parameters* i.e., parameters whose values influence the structure of the model, i.e., the set of active equations. For example, in the model `Inertia` below two equations are effectively removed from influencing the simulation if the user sets `condflg` to `false` before simulation starts.

```
model Inertia
  parameter Boolean condflg;
  ...
equation
  J*a = t1 - t2;
  if condflg then
    der(v) = a;    // Two conditional equations,
    der(r) = v;    // deactivated if condflg=false
  end if
  ...
end Inertia;
```

Variables of type `Real` are normally continuous-time variables as the default without explicit declaration of a more restrictive variability prefix. Piecewise constant (`discrete`) `Real` variables can be declared but are rarely needed. However, `Real` variables that are explicitly assigned in when-statements or implicitly assigned values through solution of equations in when-equations are also `discrete` i.e., piecewise constant.

Variables of type `Integer`, `Boolean`, `String`, or `enumeration` have data types that change value in discrete steps. Such variables may change their values only at events, which makes the default variability `discrete` natural.

Variables and expressions in functions behave as though they were discrete-time expressions since a function is always called and evaluated at some point in time i.e., time does not go forward during evaluation of a function call, similar to the evaluation of a when-equation at an event. Function calls can, however, occur at any point in time, not only at events.

For further discussion concerning variability of expressions, see Chapter 6, page 192.

3.11.3 Causality Prefixes: input and output

One of the major advantages of an equation-based language such as Modelica is that acausal mathematical models can be specified since equations do not prescribe a certain data-flow direction. However, certain model components such as functions and blocks have specified causality. When calling a function, data are supplied as actual input arguments and results are returned from the function. The same is true for blocks since a `block` is a kind of restricted Modelica class with specified causality for all variables in its interface.

The prefixes `input` and `output` are available for specifying and thus restricting the direction of data flow for declared variables. This is required for interface variables of blocks and functions, but can be used for a variable in any class e.g., to simplify the solution process when the data-flow direction is fixed and known. A simple `block` with one `input` and one `output` variable is shown below:

```
block SquareDance
  input  Real x;
  output Real squared;
equation
  squared = x*x;
end SquareDance;
```

3.11.4 Flow Prefix: flow

The `flow` prefix is required for variables which belong to instances of `connector` classes and specify flow quantities e.g., current flow, fluid flow, force, etc. Such quantities obey laws analogous to Kirchhoff's current law of summing all flows into a specific point to zero. Variables without the `flow` prefix are assumed to be potential (nonflow) quantities e.g., voltage, pressure, position, etc., which obey laws analogous to Kirchhoff's first law of setting all potential quantities connected to the same point equal. The type of variables declared with the `flow` prefix must be a subtype of `Real`. For more in-depth information concerning the `flow` prefix including examples, see Chapter 5, page 148. A small example of a `flow` variable:

```
flow Current i;
```

3.11.5 Modification Prefixes[13]: replaceable, redeclare, and final

Modification prefixes control modification of declared elements in classes. What is a modifier? It can be regarded as a kind of (incomplete) declaration that is used to change certain declarations from an inherited class. We have already seen several examples of simple modifiers that are used to change default class

[13] The reader who wants a more accessible and in-depth treatment of this topic should read Chapter 4 instead of the short presentation in this section.

attribute values. For example, the following type declaration contains three modifiers to the built-in class `Real` that modifes the attributes `unit`, `min`, and `max`.

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

The `final` prefix used at the front of a declaration or modifier prevents all modifications of a declared element and further modifications of a modified element. The motivation behind the `final` prefix is to prevent users from unknowingly introducing certain errors, which can be avoided by forbidding modifications of sensitive elements. The use of `final` also prevents interactive setting of parameter variables in the simulation environment since this is a kind of modification. The `final` prefix prohibits further modification of the `unit` attribute in the next example:

```
type Angle = Real(final unit = "rad");
Angle a1(unit="deg");          // Error, since unit declared as final!
```

The modification prefix `replaceable` used in front of a declaration enables modification in the form of a redeclaration of that declaration. The following is an example of a replaceable declaration from the class C below, of the example variable `pobj1` where `pobj1` also has a local modifier `p1=5`:

```
replaceable GreenClass  pobj1(p1=5);
```

However, modification of the default attribute *value* of a declared element is allowed also without the `replaceable` prefix, as for the `unit` attribute in the above `Voltage` declaration.

The `class C`, also presented previously in Chapter 2, Section 2.5.1, page 30.

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

When a modifier contains a complete declaration including type and/or prefixes, the `redeclare` prefix is required at the front of the modifier, which then replaces the original declaration. Such a modifier is called a *redeclaration*, and usually (but not always—see Chapter 4, page 125) requires the original declaration to be declared with the prefix `replaceable`. For example:

```
class C2 = C(redeclare RedClass pobj1);
```

The modification prefix `replaceable` when used in a modifier enables *further* modification of the resulting modified element. For example, in class `C3` this is used to enable further modification of `pobj1` when creating `C4`:

```
class C3 = C(redeclare replaceable RedClass pobj1);
class C4 = C3(redeclare YellowClass pobj1);  // OK, since replaceable
class C4 = C2(redeclare YellowClass pobj1);  // Error, not replaceable
```

Modelica classes and packages can be parameterized in a general way, corresponding to generic classes or packages (see Chapter 2, page 30, and Chapter 10, page 346). Both classes and variables which are members of some class can be used as a kind of class parameter by attaching the prefix `replaceable` in front of their declarations. It is *required* to have the prefix `replaceable` in an element declaration to allow replacing the declaration, except when only changing the default attribute value, and/or restricting the variability and possible dimension sizes.

For more details and examples regarding modifiers, see Chapter 4, page 119, regarding redeclaration and generic classes see page 125 and page 133 respectively, and on the semantics of `final` see page 128 in the same chapter.

3.11.6 Class Prefix: partial

The class prefix `partial` is used to indicate that a class is incomplete such that it cannot be instantiated. The keyword is optional i.e., a class can be incomplete without having the prefix `partial`. On the other hand the `partial` prefix is required for those partial classes for which type checking of the class declaration cannot be performed without generating type errors e.g., because a type used by the partial class might not be available where the class is declared, only where it is used. Partial classes are aimed for reuse through inheritance (see Chapter 4, page 138 for more details). The common terminology for partial class in other programming languages is *abstract* class.

3.11.7 Class Encapsulation Prefix: encapsulated

The class encapsulation prefix `encapsulated` is used to achieve lookup encapsulation of classes, i.e., the lookup stops at the class boundary. This mechanism is primarily intended for Modelica packages, since the common concept of packages entails an independent software unit used for organization of large software systems. An encapsulated package (or class) has no implicit dependencies on the declaration context within which it is placed. All dependencies are explicit via `import` statements. See Chapter 10 on packages, page 335, for more information.

3.11.8 Instance Hierarchy Lookup Prefixes: inner and outer

Lookup of names in Modelica is normally done in the usual way for lexically scoped languages with nested scopes. First the current scope containing the name reference is searched for a definition of the name; if not found the next outer scope is searched, etc.

However, *lookup through the instance hierarchy* instead of through the class nesting hierarchy is more suitable when a pervasive object property such as a *force field* or an *environment* temperature should affect all parts of the object. In Modelica such a common property can be declared as a *definition declaration* using the `inner` prefix, and subsequently accessed by *reference declarations* using the `outer` prefix. The lookup is performed through the instance hierarchy instead of through the nested class declaration hierarchy.

See Section 5.8 on page 173 for a comprehensive description of applications of this concept and Section 3.13.3 on page 99 in this chapter for a presentation of the instance hierarchy lookup mechanism.

3.11.9 Ordering of Prefixes

The *ordering* of Modelica declaration prefixes within a declaration is strictly specified according to the list below. Not all combinations of prefixes are meaningful e.g., a `constant input` variable is not very interesting since it can never change value. The modification prefix `redeclare` may only occur in modifiers, whereas the access prefixes `public` and `protected` may never occur in a modifier. The prefixes `encapsulated` and `partial` may be used only for classes, whereas `flow`, `discrete`, `parameter`, `constant`, `input`, and `output` may be used only in variable declarations.

The prefixes are placed at the beginning of a declaration element and should be used in the following order whenever more than one prefix occurs in the same declaration:

1. Access prefixes `public` and `protected`.
2. Modification prefix `final`.
3. Instance hierarchy lookup prefixes `inner` and `outer`.
4. Modification prefix `redeclare`.
5. Modification prefix `replaceable`.

6. Class encapsulation prefix `encapsulated`.
7. Class prefix `partial`.
8. The flow prefix `flow`.
9. Variability prefixes `constant`, `parameter`, and `discrete`.
10. Causality prefixes `input` and `output`.

A subset of the prefixes are denoted *type prefixes* i.e., `flow`, `discrete`, `parameter`, `constant`, `input`, `output`, and are considered part of the class type when type checking. See also Section 3.14.5 on page 104 in this chapter.

3.11.10 Variable Specifiers

After having discussed declaration prefixes in detail in the previous sections, it is useful to recall the structure of a declaration:

<prefixes> <type> <variable specifier list>

Each *variable specifier* also has a structure, where all parts apart from the identifier are optional:

<identifier> <array dimension descriptor> <modifiers> <declaration equation/initializer>

As previously stated, an array dimension descriptor can either be part of the type or be put directly after the identifier in the variable specifier. The modifiers should always be placed after the array dimension descriptor if present, otherwise directly after the identifier. More information on modifiers is available in Chapter 4, page 119. The next section shows examples of modifications of start values for specifying default initial values of variables.

3.11.11 Initial Values of Variables

A Modelica variable is either statically allocated and instantiated, which is normally the case for instantiations of models, or is dynamically instantiated as is the case for function formal parameters and local variables. The `start` value of a variable is the default initial value of the variable normally used at the beginning of a simulation. The simulation tool i.e., the solver, is, however, free (unless specifically prohibited in each declaration by setting the built-in attribute `fixed` to `true`) to choose another start value for the variable if this is necessary to be able to solve the systems of equations.

If nothing else is specified, the default start value of all statically allocated numerical variables is zero or an array containing elements of the value zero, apart from function results and local variables where the initial value is unspecified. Other start values than the default zero can be specified by setting the `start` attribute of instance variables. For example, a start value can be specified in the class `MyPoints`:

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

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

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

Note that the `start` attribute of an array variable has the same dimensionality as the variable itself. Single elements of the `start` value can be modified by indexing the attribute in the modification equation e.g., `start[2,1]` in the array `A5` example. Below are a few declarations with different combinations of array dimension descriptors and modifiers.

```
Real  A3[2,2];                        // Array variable
Real  A4[2,2](start={{1,0},{0,1}}); // Array with explicit start value
Real  A5[2,2](start[2,1]=2.3); // Array with single element modification
```

Do not make the mistake of using declaration equations for specifying initial values since this will set the variable equal to a constant value throughout the simulation. This is fine for constants or parameters since their values should be fixed during simulation, but is usually not what is expected for a normal variable. For example:

```
class test2
  parameter Real x1 = 35.2;   /* OK, since x1 has a fixed value */
  Real x2 = 99.5;             /* Probably a mistake */
end test2;
```

No equations are present in functions, not even equations for start values, since functions can contain only algorithmic code. Instead, assignment statements can be used to specify default initial values of formal parameters, function results, or local variables. Consider the example function `f2` below:

```
function f2
  input  Real x;
  input  Real y := 2;
  output Real r1;
  output Real r2 := 3;
algorithm
  r1 := 2*x;
  if r1>5 then r2 := 2*y;
end f2;
```

A call to `f2` using positional parameter passing always requires specifying values of all the arguments, which leaves the default value of `y` unused:

```
(z1,z2) := f2(33.3,44.4);
```

On the other hand, a call to `f2` using named parameter passing as below can leave out the second argument since the default is available for this argument. Note that because the conditional assignment to `r2` will never take place as a result of the call below, the default result of `3` will be used for the second result `r2`:

```
(z1,z2) := f2(x=1);
```

3.12 Declaration Order and Use Before Declaration

In Modelica, as well as in Java, variables and classes can be used *before* they are declared. This gives maximum flexibility in organizing the code to achieve logical structuring and maximum readability. For example, the well-known top-down design methodology emphasizes the major building blocks at the top level of the system structure and puts less emphasis on low-level details. A declaration before use constraint, common in several older programming languages, would force the low-level model components to be declared first, making it harder to use top-down design.

In fact, in Modelica declaration order is significant only in the following two cases:

- Declaration order is significant for input or output formal parameter declarations in *functions* with more than one input formal parameter or more than one result. The reason is that functions may be called with positional parameter passing e.g., as in the examples in the previous section.
- Declaration order is significant for the variable declarations within *record* classes. The reason is that any record can be passed as an argument to an external function, in which case the variable

ordering in the Modelica record is reflected in the variable ordering of the corresponding external record data structure.

In the example class below, which is perfectly legal, the types `Voltage` and `Current` are used before they are declared:

```
class UseBeforeDecl
  Voltage v1;
  Current i1;
  Voltage v2;
  type Voltage = Real(unit="kV");
  type Current = Real(unit="mA");
end UseBeforeDecl;
```

3.13 Scoping and Name Lookup

The scope of a name in Modelica is the region of the code where the name has a meaning corresponding to its intended use. For example, we may have declared a `Real` variable `x` in a class `RC` and an `Integer` variable `x` in another class `IC`:

```
class RC
  Real x;

  class IC
    Integer x;
  equation
     x = 2;
  end IC;

  class BC
    Real y;
  equation
    x = y;
  end BC;

equation
  x = 3.14;

end RC;
```

The scope of the `Real` variable `x` includes the whole class `RC` apart from the class `IC` where the `Real x` is hidden by a local declaration of an `Integer` variable `x` within `IC`, which has a scope that is precisely `IC`. Thus, scopes usually correspond to class declarations.

3.13.1 Nested Name Lookup

Modelica uses nested lookup of names like most other programming languages. The lookup procedure searches for declarations of names within local nested scopes before going out to enclosing scopes. This is the reason why the `Real x` is hidden within `IC`—the search procedure finds the declaration of the integer `x` first: it does not even need to investigate the declarations within `RC`.

However, even if there was no local declaration of the variable x, it would still be illegal to refer to the `Real x` in the enclosing scope `RC`—*references* to *variables and parameters in enclosing scopes is illegal* in Modelica—such nested lookup is only allowed for names of constants and declared classes. For example:

```
package A
  constant Real z = 2;
```

```
model B
  Real z;
  function foo
    output Real y
  algorithm y := z; // Illegal since reference to non-constant z in B
  ...
```

It is not only classes that introduce declarations with local scopes. The iteration variable in Modelica for-loops has a scope that extends throughout the for-loop, and hides declarations of variables with the same name in enclosing scopes. For example:

```
function PolynomialEvaluator3
  input  Real a[:];
  input  Real x;
  output Real y;
protected
  Real   xpower;
  Integer i;  (* Declared local variable i *)
algorithm
  y := 0;
  xpower := 1;
  for i in 1:size(a, 1) loop  (* Iteration variable i *)
     y := y + a[i]*xpower;
     xpower := xpower*x;
  end for;
end PolynomialEvaluator3;
```

The iteration variable `i` introduced by the for-loop in the body of the function `PolynomialEvaluator3` has a scope that extends throughout that for-loop, thus preventing access to the declared local variable `i` from within the for-loop.

3.13.2 Nested Lookup Procedure in Detail

The following stepwise procedure specifies the semantics of lookup for a name used within some piece of Modelica code. The lookup starts in the innermost scope containing the code, usually a class, and proceeds through enclosing scopes until the outermost scope or an encapsulated package or class is encountered. The notion of class should be interpreted in the general sense, i.e., it can also be a function or package.

Certain language constructs referred to below like `import` statements, `encapsulated` packages, and `encapsulated` classes are explained in more detail in Chapter 10. To summarize, the statement

```
import A.B.Myclass;
```

makes it possible to access `Myclass` using the short name `Myclass` instead of `A.B.Myclass`. When importing a whole package

```
import A.B;
```

makes it possible to use the name `B.Myclass` with a short package prefix `B` for access, whereas

```
import A.B.*;
```

allows the use of short names such as `Myclass` for all classes in `A.B`.

The lookup procedure is as follows, excluding lookup of names with the prefix `outer`, which is described in Section 3.13.3, page 99:

1. First look for implicitly declared iteration variables in possible enclosing for-equations or for-statements, if the analyzed code is inside the body of a for-construct. Regarding for-equations, see Section 8.3.2, page 241, and for-statements, see 9.2.5, page 288.

2. When a piece of Modelica code is analyzed, e.g. a class or variable declaration, an `extends` clause, an equation or algorithm, encountered names in the code are looked up. Each such name is looked up in the sequence of enclosing scopes starting at the innermost scope. The search continues until a match is found or the outermost scope or a scope corresponding to an encapsulated class or encapsulated package has been reached. (However, even if found, it is illegal to reference a variable or parameter declaration in an enclosing scope, whereas constants and classes can be accessed—similar to the rules for definitions in packages).

   If not found the lookup continues in the global scope containing the predefined Modelica types, functions, and operators. (For example, the predefined function `abs` is searched upwards in the hierarchy as usual. If an encapsulated boundary is reached, `abs` is searched for in the global scope instead. The operator `abs` cannot be redefined in the global scope, because an existing class cannot be redefined at the same level.) Certain operators are denoted by reserved words or special characters and therefore do not need lookup (see the Modelica Grammar in Appendix A).

The lookup in each scope is performed in the following order until a match is found:

- Search among *declared* named elements i.e., *classes and variables* declared in the scope, including elements inherited from other classes. Note that import statements are not inherited from other classes. Regarding inheritance see Chapter 4, and concerning `import` see Chapter 10, page 335.
- Search among the import names of *qualified import statements* in the scope. Such an import name may refer only to an imported package, class, or constant. The *import name* of the qualified import statement `import A.B.C;` is `C` and the import name of the *renaming* qualified import statement `import D=A.B.C;` is `D`. In both cases `A.B` must be a package. Regarding the special case `import E;`, `E` can be any class or package as an element of the unnamed top-level package.
- Search among the public members of packages imported via *unqualified import statements* in the current scope e.g., `import A.B.*;`. It is an error if this step produces matches from several unqualified imports. Unqualified import statements may only import from packages i.e., `A.B` must be a package in the statement `import A.B.*;`.

Lookup of the name of an imported package or class e.g., `A.B.C` in the statements `import A.B.C; import D=A.B.C; import A.B.C.*;`, deviates from the normal lookup by starting the lookup of the first part of the package/class name at the top level of the hierarchy.

The lookup procedure for *qualified names* i.e., names like `C.x` and `A.B.D.y` with dot notation, is a somewhat more involved procedure. The lookup for the first part of a name i.e., `C` in `C.x`, or `A` in `A.B.D.y`, proceeds as described above, starting with the current scope. If no matching variable declaration is found, the first part of the name is assumed to be a package name.

For a qualified name of the form `A.B`, or `A.B.C`, etc., lookup is performed as follows:

1. The first part of the name e.g., `A`, is looked up as defined in step 1 above.
2. If the first part of the name matches the name of a *declared variable*, the rest of the name e.g., `B` or `B.C`, is looked up among the named attributes (variables) of that variable, since the variable must be an instance of a record or other kind of class.
3. If the first part of the name denotes a class, typically *a package*, that class declaration is analyzed and temporarily expanded i.e., performing inheritance and modifications of the class in the lookup context of the hierarchy of enclosing classes of that class. The rest of the name e.g., `B` or `B.C`, is looked up among the declared named elements of that temporarily expanded class declaration.

If the class is not a package, the lookup of the rest of the name is restricted to elements that are encapsulated packages or encapsulated classes. Such lookup can be used, e.g. to access a local version of the Modelica library embedded within a `model` restricted class that is not a package. See also Chapter 10, page 353.

The top level of the package/class hierarchy referred to above can be viewed as an unnamed package containing declared elements loaded into the current workspace together with packages stored in the file system at positions specified by MODELICAPATH. See Chapter 10, page 344 for more information on MODELICAPATH.

3.13.3 Instance Hierarchy Lookup Using inner and outer

As we already stated briefly in Section 3.11.8, standard nested lookup as described in the previous sections is not convenient in some situations where a pervasive object property should be accessible to all components of the object. For example:

- The components of a planetary system influenced by a common gravitational field.
- Electrons influenced by an electromagnetic field.
- An environmental temperature or pressure influencing the components of the system in question.

Section 5.8 in Chapter 5, page 173 contains a more thorough discussion of such applications, including the modeling of gravitational fields.

As an example, the temperature of a circuit board (see the example below) should be accessible to all components on the board even though the classes of those components are not declared inside i.e., nested inside, the class of the board itself.

Regarding the circuit board model, it is not practical to place the class definitions of the components inside a specific board class, since they should be usable also for models of other boards. Rather, the common property belongs to the object which contains the components in question. This leads us to the concept of *lookup within the instance hierarchy*.

Based on these considerations, the `inner`/`outer` concept has been introduced in Modelica to provide access to common properties by specific *definition* and *reference* declarations, with lookup through the instance hierarchy that *overrides* the usual nested lookup:

- A declaration with the prefix `inner` is a *definition declaration* of an element that *defines* an object property or element in common to the components of the object, and therefore should be accessible from within those components.
- A declaration with the prefix `outer` is a *reference declaration* of an element that *references* a common property or element defined by a definition declaration in the enclosing instance hierarchy.

Note that the enclosing instance hierarchy is the hierarchy of parts/instances that the current instance is a member of.

The following holds for an *instance hierarchy lookup* of the name of a reference declaration:

- The search for a matching *definition* declaration (`inner`) with the same name as in the *reference* declaration (`outer`) is from the current instance to containing instances within the *instance hierarchy* of the reference declaration, not necessarily within the *class nesting hierarchy*.
- The definition declaration (`inner`) can be shadowed by nested declarations with the same name.
- The item declared in the definition declaration (`inner`) should be a subtype (see Section 3.14, page 100 in this chapter) of the type of the corresponding reference declaration (`outer`) or a subtype of an explicitly given constraining type. Also, the reference declaration should not have any modifications.

Note that instance hierarchy lookup is more selective than conventional lookup of global variables found in many other programming languages, since the search is only in the hiearchy starting at a specific instance.

We give a simple example in terms of an electric circuit board `CircuitBoard1` containing temperature-dependent components that depend on a common board temperature.

The two kinds of hierarchies, the usual *class nesting hierarchy* as well as the *instance hierarchy*, are depicted in Figure 3-8 for the `BoardExample` class shown in the following.

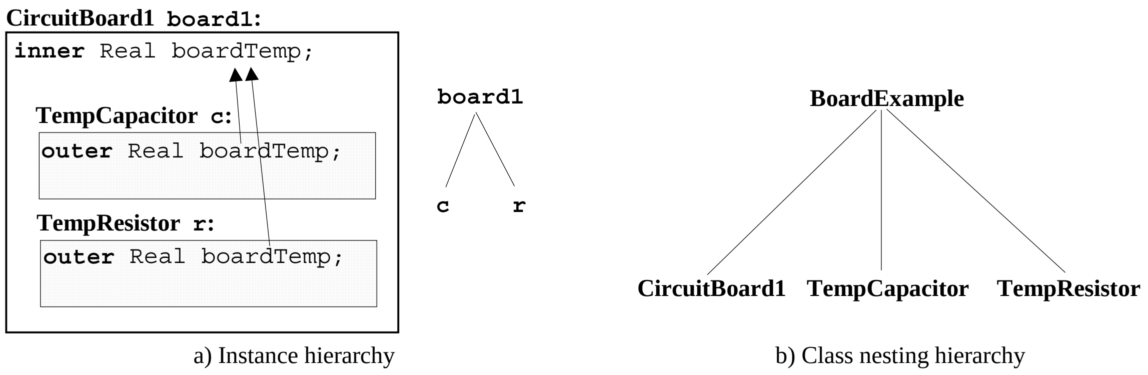


Figure 3-8. Instance hierarchy (a) and class nesting hierarchy (b) for the electric circuit board example.

All components in the electric circuit board should reference exactly the same board temperature variable `boardTemp`, which is defined by an `inner` declaration within the `CircuitBoard1` class. Each component contains an `outer` reference declaration referring to the board temperature `boardTemp`, which is found during the *instance hierarchy lookup* to be the `inner` board temperature `boardTemp` of `CircuitBoard1` since the components `c` and `r` belong to `board1`.

```
class BoardExample

  class CircuitBoard1
    inner Real BoardTemp;  // Definition of the board temperature variable
    TempCapacitor c   "Component within CircuitBoard1 instance";
    TempResistor  r   "Component within CircuitBoard1 instance";
  end CircuitBoard1;

  class TempCapacitor
    outer Real BoardTemp;  // Reference to the board temperature variable
    Real T;
  equation
    T = BoardTemp;
  end TempCapacitor;

  class TempResistor
    outer Real BoardTemp;  // Reference to the board temperature variable
    Real T;
  equation
    T = BoardTemp;
  end TempResistor;

  CircuitBoard1 board1  "CircuitBoard1 instance";
end BoardExample;
```

For additional examples on how to use this facility see Section 5.7.5.5 in Chapter 5, page 172.

3.14 The Concepts of Type and Subtype

What is a type? This is a relevant question in the context of programming languages. We are talking about the type of variables, of values, of classes, etc., making it important to convey some intuition about the concept.

A *type* can conceptually be viewed as a *set of values*. When we say that the variable `x` has the type `Real`, we mean that the value of `x` belongs to the set of values represented by the type `Real` i.e., roughly the set of floating point numbers representable by `Real`, for the moment ignoring the fact that `Real` is also viewed as a class with certain attributes. Analogously, the variable `b` having `Boolean` type means that the value of `b` belongs to the set of values {`false`, `true`}. The built-in types `Real`, `Integer`, `String`, `Boolean` are considered to be distinct types.

The *subtype* relation between types is analogous to the subset relation between sets. A type `A1` being a subtype of type `A` means that the set of values corresponding to type `A1` is a subset of the set of values corresponding to type `A`.

Thus the type `TempResistor` below is a subtype of `Resistor` since the set of temperature-dependent resistors is the subset of resistors having the additional variables (attributes) `RT`, `Tref` and `Temp`. The type `Integer` is not a subtype of `Real` in Modelica even though the set of primitive integer values is a subset of the primitive real values since there are some attributes of `Real` that are not part of `Integer` (see Figure 3-9).

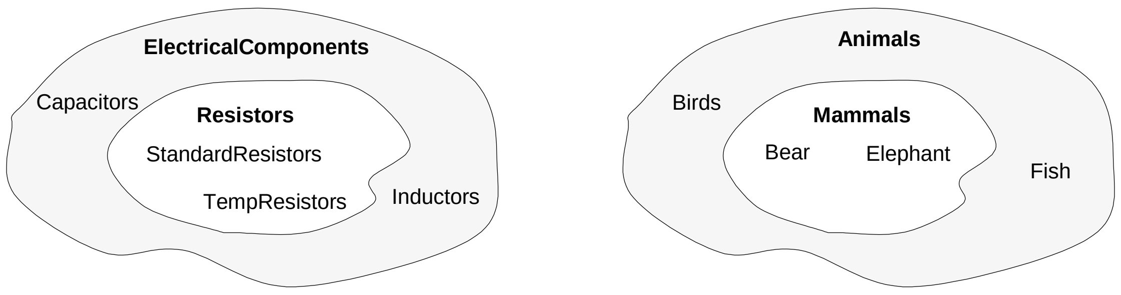


Figure 3-9. Relation between the concepts of subtype and subset in the domains of electrical components and animals.

The *supertype* relation is the inverse of the subtype relation. Since `TempResistor` is a subtype of `Resistor`, then `Resistor` is a supertype of `TempResistor`.

```
model Resistor  "Electrical resistor"
  Pin p, n  "positive and negative pins";
  Voltage v;
  Current i;
  parameter Real R(unit="Ohm")           "Resistance";
equation
  v = i*R;
end Resistor;

model TempResistor "Temperature dependent electrical resistor"
  Pin p, n  "positive and negative pins";
  Voltage v;
  Current i;
  parameter Real R(unit="Ohm")           "Resistance at reference Temp.";
  parameter Real RT(unit="Ohm/degC")=0   "Temp. dependent Resistance.";
  parameter Real Tref(unit="degC")=20    "Reference temperature.";
  Real           Temp=20                 "Actual temperature";
equation
  v = i*(R + RT*(Temp-Tref));
end TempResistor;
```

The definitions of `Pin`, `Voltage`, and `Current` that are used in the resistor classes above were previously defined in Section 2.8.3, page 38.

3.14.1 Conceptual Subtyping of Subranges and Enumeration Types

Conceptually, a subrange is a subset of an ordered set of elements, defined as the ordered set of all elements from the first to the last element in the subrange. A number of programming languages e.g., Ada, Pascal, etc., allow the user to define subrange types of enumerable types, for example integer, subranges which appear as follows in Pascal:

```
2..10
```

This subrange is conceptually a subtype of `Integer` and also a subrange type of integers from 2 to 10. The general idea of subrange subtypes as subsets of ordered enumerable elements is illustrated in Figure 3-10, where `SmallSizes` is a subtype of `AllSizes`, `Int_1_2` a subtype of `Int_1_4`, and `FalseBoolean` a subtype of `Boolean`.

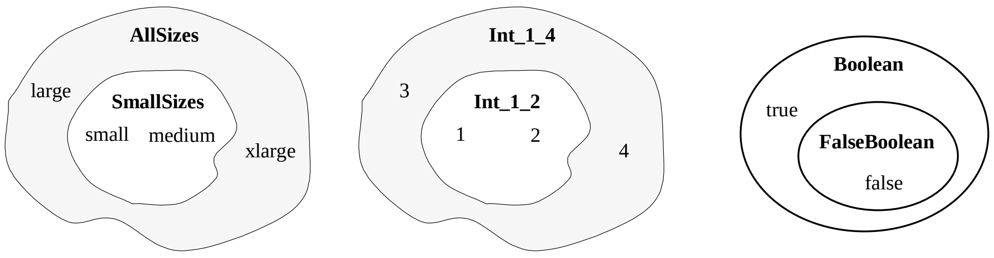


Figure 3-10. Conceptual subtyping examples of enumeration types, integer ranges, and boolean ranges. However, general subrange subtyping as illustrated in this figure is not in the current version of Modelica.

In Modelica we can define enumeration types e.g., as follows:

```
type AllSizes   = enumeration(small, medium, large, xlarge);
type SmallSizes = enumeration(small, medium);
```

According to the examples in Figure 3-10 and general subset relations, `SmallSizes` should be a subtype of `AllSizes`. However, this is not the case in the current version of Modelica, which does not support definition of subrange subtypes.

Instead, subtyping of enumeration types usually requires type identity, see Section 3.14.9, page 107, based on the fact that two identical sets are subsets of each other. This restrictive subtyping rule might be extended in the future, but that will require enhancements in several places of the Modelica language. There is also a special enumeration type `enumeration(:)`, which is a supertype of all possible enumeration types. For example:

```
S1 = enumeration(small,medium);        // S1 not subtype of S2
S2 = enumeration(small,medium,large);  // S2 is not a supertype of S1
S3 = enumeration(small,medium,large);  // S3 is a subtype of S2
S4 = enumeration(:);                   // S1, S2, S3 are subtypes of S4
```

3.14.2 Tentative Record/Class Type Concept

A *record/class type* in Modelica is conceptually the set of all possible record values of that record type, where each value contains named values for the variables (also known as fields or attributes) of that record. Thus, both the *record field names* and the *record field types* are significant when defining a record type. Duplicate field names in a type are not allowed. In general, a tentative definition of the type of a record/class with N elements can be given by a descriptor that is a set of pairs of the form (*field-name*, *field-type*) where field ordering is unspecified, analogous to several type systems for object-oriented languages by Luca Cardelli et al. [Abadi-Cardelli-96]:

(*name1*:type1, *name2*:type2, ... *nameN*:typeN)

We will gradually develop this tentative type concept into a more general type concept suitable for all type aspects of the Modelica language. However, returning to the view of a record type as a set of record values, consider the `Person` record declaration below:

```
record Person
  String name;
  Real   weight;
  Real   length;
end Person;
```

A set of `Person` values might appear as follows, using a pseudosyntax for record values (not conforming to Modelica syntax), where both the field name tag and the field data value are part of a `Person` value:

```
{(name:"Eve", weight:65, length:170),
(name:"Adam", weigth:80, length:180), etc. }
```

In general the ordering between the variables of a class is not considered significant except in two cases:

1. The ordering within the two sets: function formal parameters and function results.
2. The ordering between the variables of a record being passed to an external function.

However, except for the case of function types, the ordering between variables (fields) is not considered significant for type checking. Therefore the following record value is also considered compatible with type `Person`:

```
(weight:65, length:170, name:"Carl")
```

3.14.3 Array Type

An *array type* is conceptually a set of array values, where each array value can be considered as a *set* of pairs (*array-element-index*, *array-value*) analogous to the set of pairs (record-field-name, record-field-value) that make up a record value. Thus, for a vector type with four elements, with the possible array indices '1', '2', '3', '4', where the array index is used as a name instead of the field name in a record, a set of array values with labeled array elements could appear as follows in pseudocode:

```
{ ('1':35.1, '2':77, '3':38, '4':96),
  ('1':5.1,  '2':11, '3':305,'4':83), etc... }
```

Following our informal definition of array type, each possible index value for an array should also be part of the array type in the same way as all variable names of a record are significant. However, this is impractical when there is a large number of array elements and in cases where the number of array elements is unknown. Instead we will soon turn to a more compact type representation.

Unspecified dimension sizes is generally the case for library functions operating on arrays. For example, the library function for inverting a matrix should work on any size of matrix, and therefore has the dimension sizes unspecified, causing the number of array elements to be unspecified. It would not make sense to have a separate inverting function for 2 × 2 matrices, another for 3 × 3 matrices, etc. Thus it must be possible to define array types for which certain array dimension sizes are unspecified. Two vector types in Modelica are shown below:

```
type Vec2 = Real[2];   // Vector type with specified dimension size
type Vec  = Real[:];   // Vector type with unspecified dimension size
```

We now replace our informal set definition of *array type* above with a more compact type descriptor still with the same form as the one used for record types. This descriptor is a set containing a single pair (*index-specifier*, *element-type*), which can be used since all array elements share the same element type in contrast to records where variables (fields) usually have different types. In this array type representation the *element-type* is the type of the array elements and the *index-specifier* represents all possible legal indices for the array where each dimension in an index is represented by a nonnegative size expression for the maximum index value of that the dimension, or a colon (:) representing unspecified dimension size:

```
( (index-specifier, element-type))
```

The array type descriptors for the two types `Vec2` and `Vec` would then become the following:

```
( ('2', Real) )
( (':', Real) )
```

Turning to the problem of representing multidimensional arrays we observe that in Modelica as well as in many programming languages a multidimensional array is defined as an array of arrays. Thus a 2 × 3 matrix type is viewed as a 2-sized vector of 3-sized vectors. For example:

```
type Mat23 = Real[2,3]; // Matrix type with specified dimension size
type Mat   = Real[:,:]; // Matrix type with unspecified dimension size
```

These two matrix types will have the following type representation given the array of arrays representation:

```
( ('2', (('3', Real)) ) )
( (':', ((':', Real)) ) )
```

The subtype relation between two array types, A subtype of B, requires that the element-type of A is a subtype of the element-type of B and each corresponding dimension specifier in the index-specifier is either identical size or the dimension specifier of B is a colon (:). Thus, `Mat23` is a subtype of `Mat`. This is reasonable since arrays of type `Mat23` can be used in places where type `Mat` is required, but not vice versa.

3.14.4 Function Type

The definition of a *function type* is analogous to our definition of general class type above with the additional properties that the *ordering* between the fields (variables) i.e., the formal parameters of the function, is significant and that the prefixes `input` and `output` for each formal parameter are also part of the function type. A function with three input formal parameters and one result is actually quite different from a function with two input formal parameters and two results, even if the names and types of the formal parameters are the same in both cases.

Two function types that are subtypes of each other also need to fulfill type equivalence as defined further below. This is more restrictive than subtyping for general class types. The reason is to avoid dealing with certain problematic issues regarding substitutability of functions (see Chapter 9 for more details on functions).

3.14.5 General Class Type

Regarding function types we concluded in the previous section that the prefixes input and output are also part of the type. In fact, all *type prefixes* when present in the variable declarations within a class declaration i.e., `flow`, `discrete`, `parameter`, `constant`, `input`, `output`, as well as the `final` element prefix, are considered part of the class type not only for functions but for all kinds of classes in Modelica. The variability prefixes `discrete`, `parameter`, `constant` are considered part of the type since they influence the semantics and type checking of assignment statements and equations. The `flow` prefix is part of the type since it affects the semantics of connection equations. Finally, the `final` prefix influences the semantics of inheritance and modifications, described in more detail in Chapter 4.

Thus, our tentative definition of a *class type* is *generalized* to be a sequence of pairs (*element-name*, *element-prefixes&type*) for all N public elements in the class, where *element ordering* is part of the type but not used in type checking except in certain situations such as regarding functions, and *index-specifier* is used instead of element-name for array types:

(*name1*: prefixes1&type1, *name2*: prefixes2&type2, ... *nameN*: prefixesN&typeN)

Given the simple example function `multiply` below:

```
function multiply;
  input  Real x;
  input  Real y;
  output Real result;
algorithm
  result := x*y;
end multiply;
```

Using our general definition of type to represent this function we obtain the following:

(*x*: input&Real, *y*: input&Real, *result*: output&Real)

3.14.6 Types of Classes That Contain Local Classes

So far we have dealt primarily with classes that only contain attributes or fields (variables) that can contain data ("data fields") i.e., similar to a classical record type. However, in Modelica it is possible to declare local classes inside a class, as, for example, the classes `IC` and `BC` inside the class `RC2` below.

```
class RC2
  Real x;     // field x
  class IC
    Integer x;
  equation
    x = 2;  // Refers to local x inside IC
  end IC;

  class BC
    Real y;
  equation
    x = y;  // Refers to x in RC2
  end BC;
  IC ix;      // field ix, instance of IC
  BC bx;      // field bx, instance of BC

equation
    x = 3.14;
end RC2;

...
RC2  z;
...
```

The three variables ("data fields") of class `RC2` are `x`, `ix`, and `bx`. There are also two local classes (not "data fields") `IC` and `BC` inside `RC2`. They can be referred to only within the local scope of `RC2`, but in principle they could be moved outside `RC2` with some changes of names. An outside class `BC` could appear as follows:

```
class RC2_BC   // Hypothetical BC outside RC2
  Real y;
equation
  z.bx.x = y;  // Refers to x in the z instance of RC2
end RC2_BC;
```

A version of RC2 referring to such external classes would appear as follows:

```
class RC2_v2   // Hypothetical RC2 referring to outside IC and BC
```

```
    Real x;       // field x

    RC2_IC ix;    // field ix, instance of RC2_IC
    RC2_BC bx;    // field bx, instance of RC2_BC
  equation
    x = 3.14;
  end RC2_v2;
```

From this perspective it would be natural to include only the three "data fields" `x`, `ix`, and `bx` in the type of `RC2`. Operations on and use of data objects of type `RC2` involve only these three "data fields".

On the other hand the class `RC2_BC` is not very general since it refers only to the `z` instance of `RC2` in its equation `z.bx.x = y`. The hypothetical notation `RC2.bx.x` in that equation is not possible since that would imply that `bx` is a class variable (see page 74), which is not the case. Also, class variables are not currently allowed in Modelica. Furthermore, one might argue that a class containing local classes is different from a class without local classes, and that the local types `IC` and `BC` need to be included anyway since they are referred to by `ix` and `bx`. These considerations have led to the design decision that in Modelica local classes are considered as elements of the type of the class within which they are declared. The "*field name*" of such a local class is the class name, and the type is the type representation of the local class as a set of pairs according to our general type representation, but for brevity denoted by names such as `RC2.IC` and `RC2.BC`. The general type representation of class `RC2` then becomes as follows:

(*x*: Real, *IC*: RC2.IC, *BC*: RC2.BC, *ix*: RC2.IC, *bx*: RC2.IC)

3.14.7 Subtyping and Type Equivalence

Having defined the notion of type for the different kinds of classes and objects that can occur in Modelica, we need to turn to the issue of how and when types are used.

Type equivalence is needed to ensure compatibility of data and objects in different situations such as equality between the left-hand side and the right hand-side in equations, ensuring that the types of all arguments to the array constructor are equivalent, etc.

We have informally defined the notion of *subtype* as analogous to the notion of subset e.g., with the example of `TempResistor` being a subtype of `Resistor`. The notion of subtype is usually associated with the idea of specialization and substitutability: if T2 is a subtype of T1, then an object of T2 should usually be allowed to be substituted instead of an object of T1, or assigned to a variable of type T1 e.g., a `TempResistor` being substituted for a `Resistor`.

Below we will discuss the use of subtyping and type equivalence, and provide more precise definitions of those notions.

3.14.8 Use of Type Equivalence

Type equivalence has the following main uses:

- The types of the left- and right-hand sides of equations (e.g., $x = y$) must be equivalent after first doing type coercion, e.g. an `Integer` expression at one side of an equation is automatically converted to a `Real` expression if the other side has type `Real`.
- *Assignment statements*. In an assignment statement e.g., *a* := *expr*, the type T2 of the right-hand side must be equivalent to the type T1 of the left-hand side after possible type coercions. For example, type conversions makes assignments such as `Real[:] vec := {1,2}` possible, where the right-hand side has type `Real[2]`and the left side has type `Real[:]`, where the type of the left-hand side is coerced into `Real[2]`. Additionally, the standard coercion rule of converting an `Integer` expression to a `Real` expression when it occurs at the right-hand side of an assignment to a `Real` variable is automatically applied in order to satisfy type equivalence.

- *Argument passing at function calls*. The type T2 of an argument passed to a formal parameter (of type T1) of some function needs to be equivalent to the type T1 after possible type coercions. Passing a 2D array where a 1D array is expected is not allowed. However, a 1D array slice e.g., a row or column, can be extracted from the 2D array and passed in that case.
- The types of the arguments to the array construction function `{}` as well as to the relational operators (==, <>, <=, >=, <, >) must be equivalent. `Integer` arguments are converted to `Real` arguments when needed to achieve type equivalence. Relational operators are defined only for scalar arguments, i.e., not for arrays and user-defined class types.

3.14.9 Definition of Type Equivalence

Two types T1 and T2 are *equivalent* if they have the same named public elements and if the maximally expanded types (where type names have been replaced by corresponding type representations) of corresponding elements are the same. The elements of these types can be either variables or local classes. This can be expressed more formally as follows:

- T1 and T2 denote the same primitive type i.e., one of `'RealType'`, `'IntegerType'`, `’StringType'`, `'BooleanType'`, or `'EnumType'`. Note that these primitive types cannot occur in any context other than being part of the corresponding predefined types `Real`, `Integer`, `String`, `Boolean`, and enumeration types.
- T1 and T2 are classes containing the same public declaration elements (according to their variable names for variables and class names for local classes). The types of corresponding elements with the same names in T1 and T2 need to be equivalent.

This definition does not require any correspondence of the prefixes of the elements of the two types T1 and T2. Neither is the same ordering of the elements required. It is the same as requiring that the type representations of T1 and T2 are identical when represented as sets of pairs according to our tentative type definition above and where the element types have been maximally expanded to their primitive type components.

3.14.10 Use of Subtyping

- Subtyping has the following uses, assuming that T2 is a subtype of T1:
- *Redeclarations*. The type of a class member can be changed through a redeclaration when the class is inherited through a modification. In a `replaceable` declaration there can be an optional constraining type. In a redeclaration of a `replaceable` element the type T2 of the variable must be a subtype of the constraining type T1. Furthermore, the constraining type of a `replaceable` redeclaration i.e., a redeclaration that can be further redeclared, must be a subtype of the constraining type of the declaration it redeclares. For more details and explanations of redeclaration, modification, and inheritance see Chapter 4, page 111.
- *Instance hierarchy lookup*. The type of a definition declaration with prefix `inner` must be a subtype of the type of the corresponding reference declaration with prefix `outer` that references the definition declaration.

A class T2 that inherits a class T1 naturally becomes a subtype of T1. However, inheritance is not required for subtyping. In one of our previous examples `TempResistor` is a subtype of `Resistor` even though it does not inherit `Resistor`.

3.14.11 Definition of Subtyping

A type T2 is a subtype of a type T1 if and only if:

- Type T2 contains all the public elements with the same names as in the set of public elements in T1.
- The types of elements in T2 are subtypes of corresponding elements with the same names in T1.
- The variability of type T2 is lower or equal to the variability of type T1. The degrees of variability are the following, from the lowest variability to the highest: *constant*, *parameter*, *discrete-time*, *continuous-time*, see also page 89 on variability prefixes in this chapter, and variability of expressions in Chapter 6, page 192. However, not that the variability of a function call (not a function type) is dependent on the variability of its actual arguments.
- Additionally, for *array types* T1 and T2, the array element type of T2 must be a subtype of the element type of T1, the number of dimensions must be equal for both types, each array dimension size value must be the same in T2 and T1 (i.e., `size(T2)=size(T1))`, or the corresponding dimension size in T1 is unspecified, denoted by (:).
- Additionally, for *function types* T1 and T2, the types must have the same number of formal parameters (seen as class attributes) prefixed by `input` and `output` respectively, and keep the same relative ordering of formal parameters within each of the two groups of `input` and `output` parameters. Additionally, type equivalence is also required between T1 and T2.

This definition of subtyping fits well with the general class type definition given earlier in Section 3.14.5 on page 104 in this chapter. All the information needed to determine the subtyping relation is present in the type, for all possible kinds of types in Modelica, and where elements of a type can be either variables or local classes.

Furthermore, we should not forget subtyping rules for enumeration types, which are a kind of subrange types that were briefly discussed earlier in Section 3.9.3 page 86. Currently the Modelica enumeration subtyping rules are rather restrictive since in most cases type equivalence is required. However, this might be generalized in the future.

For any *enumeration types* S and E, S is a supertype of E and E is a subtype of S if and only if:

- The types S and E are equivalent, or
- S is the type `enumeration(:)`.

Below are some examples of enumeration subtyping:

```
E1 = enumeration(one,two,three);       // E1 not a subtype of E2
E2 = enumeration(one,two,three,four);  // E2 is not a supertype of E1
E3 = enumeration(one,two,three,four);  // E3 is a subtype of E2
E4 = enumeration(:);                   // E1, E2, E3 are subtypes of E4
```

3.15 Summary

This chapter has its focus on the most important structuring concept in Modelica: the class concept. We started by a tutorial introduction of the idea of contract between designer user together with the basics of Modelica classes as exemplified by the moon landing model, but excluding inheritance which is presented in Chapter 4. We continued by presenting the idea of restricted classes, the predefined types in Modelica, and the structure of declarations—both class declarations and variable declarations. We concluded by a presentation of the two kinds of lookup in Modelica: nested lookup and instance hierarchy lookup, followed by a description of the typing and subtyping concepts of Modelica and their usage.

3.16 Literature

The primary reference document for this chapter is the Modelica language specification (Modelica Association 2003). Several examples in this chapter are based on similar examples in that document, and in (Modelica Association 2000). Some formulations from the Modelica specification are reused in this chapter in order to state the same semantics. In the beginning of this chapter we mention the idea of a contract between software designers and users. This is part of methods and principles for object-oriented modeling and design of software described in (Rumbaugh 1991; Booch 1991; Booch 1994; Meyer 1997). The moon landing example is based on the same formulation of Newton's equations as in the lunar landing example in (Cellier 1991). The typing and subtyping concepts of Modelica are inspired by some of the simpler type systems described in the introductory part of (Abadi and Cardelli 1996).

3.17 Exercises

Exercise 3-1:

Creating a Class: Create a class, Multiply, that calculates the product of two variables, which are `Real` numbers with given values.

Make the previous class into a model.

Exercise 3-2:

Creating a Function and Making a Function Call: Write a function, `AddTen`, that adds 10 to the input number. Make a function call to `AddTen` with the input `3.5`.

Exercise 3-3:

Creating a Local Class: Here you should test that local classes are possible. For example, given the trivial class `Add` below, create a new empty class `AddContainer`, and place `Add` as a local class inside `AddContainer`.

```
class Add
  Real x = 1;
  Real y = 2;
  Real z;
equation
  x + y = z;
end Add;
```

Exercise 3-4:

Creating Instances: Consider the class `Student` below.

```
class Student
  constant  Real   legs = 2;
  constant  Real   arms = 2;
  parameter Real   nrOfBooks;
  parameter String name = "Dummy";
end Student;
```

Create an instance of the class `Student`.

Create another instance and give the student the name `Alice`. This student has 65 books.

You also want to create a student called `Burt` who owns 32 books.

Exercise 3-5:

Question: What is a variable?

Question: Which are the two kinds of variables and what is the difference between them?
Question: What is a parameter?

Exercise 3-6:

Declaring Variables: Define a class `Matrices` that declares the variables `m1`, `m2`, `m3`, `m4` all being matrices with `Integer` respectively `Real` numbers. Make at least one of them an array. Let `m1` and `m2` be parameters, `m3` a regular variable and, `m4` a constant.

Exercise 3-7:

Question: How can a variable be initialized?

Initializing Variables: Declare the variables `length1`, `length2`, and `length3` of type `Real` with different initial values. One of the variables should have the value 0. Name the class `Variables`.

Exercise 3-8:

Creating a Record: Create a record, `Dog`, with the values `nrOfLegs`, which is an `Integer` and name, which is a `String`. Give the variables some values.

Exercise 3-9:

Question: What is a type?
Question: What is a subtype?
Question: What is a supertype?

Subtype and Supertype:

```
class House
  Real length;
  Real width;
end House;
```

Define a class `Room` that is a subtype of the class `House`. This also means that `House` is a supertype of `Room`. Also give some values to the variables.

Exercise 3-10:

Access Control: Complete the classes `Access` and `AccessInst` below by adding simple equations. The variable `acc` is an instance of the class `Access`, which contains the variables `a`, `b`, `c`, `d` and `e`, of which `d` and `e` are protected. Where can the variables be referenced e.g., in equations setting them equal to some numeric constant values? (Set `a = 7.4` in the class `AccessInst`.)

```
class Access
  ...
equation
  ...
end Access;

class AccessInst
  Acc acc;
end AccessInst;
```

Exercise 3-11:

Introduce a ground force into the classes of the `MoonLanding` model to make the lander stand still after the landing, thus avoiding accelerating toward the center of the moon.