Chapter 3
Modelica: Equation-Based, Object-Oriented Modelling of Physical Systems

Peter Fritzson

Abstract The field of equation-based object-oriented modelling languages and tools continues its success and expanding usage all over the world primarily in engineering and natural sciences but also in some cases social science and economics. The main properties of such languages, of which Modelica is a prime example, are: acausal modelling with equations, multi-domain modelling capability covering several application domains, object-orientation supporting reuse of components and evolution of models, and architectural features facilitating modelling of system architectures including creation and connection of components. This enables ease of use, visual design of models with combination of lego-like predefined model building blocks, ability to define model libraries with reusable components enables. This chapter gives an introduction and overview of Modelica as the prime example of an equation-based object-oriented language.

Learning Objectives

After reading this chapter, we expect you to be able to:

- Create Modelica models that represent the dynamic behaviour of physical components at a *lumped parameter* abstraction level
- Employ Object-Oriented constructs (such as class specialisation, nesting and packaging) known from software languages for the reuse and management of complexity
- Assemble complex physical models employing the component and connector abstractions
- Understand how models are translated into differential algebraic equations for execution

3.1 Introduction

Modelica is primarily a modelling 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 behaviour 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 modelling 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.

Peter Fritzson
Linköping University, Department of Computer and Information Science, SE-58183 Linköping, Sweden
e-mail: peter.fritzson@liu.se

P. Carreira et al. (eds.), *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-43946-0_3

- Modelica has multi-domain modelling 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 modelling and simulation. We will start with a gentle introduction to Modelica from the very beginning.

3.2 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 which for example can be 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 behaviour 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 HelloWorld Modelica program solves a trivial *differential equation*:

$$\dot{x} = -a.x \tag{3.1}$$

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[1], or open the DrModelica electronic document (part of OpenModelica) containing most 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 OpenModelica environment produces a solution between time 0 seconds and time 2 seconds:

simulate(HelloWorld,stopTime=2)[2]

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

[1] There is a downloadable open source Modelica environment OpenModelica from OSMC at www.openmodelica.org, Dymola is a product from Dassault Systems www.dymola.com, see www.modelica.org for more products.

[2] Before simulating an existing model, you need to open it or load it, e.g. loadModel(HelloWorld) in OpenModelica. The OpenModelica command for simulation is simulate. The corresponding using Dymola is simulateModel("HelloWorld", stopTime=2).

plot(x)[3]

(or the longer form plot(x,xrange={0,2}) that specifies the x-axis), giving the curve in Figure 3.31:

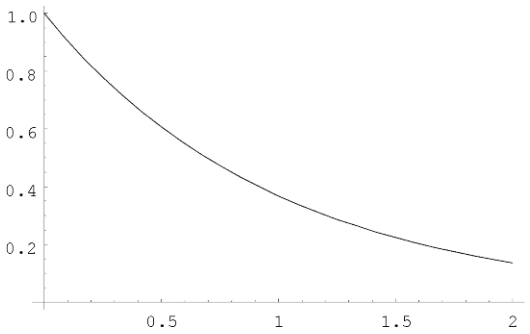

Fig. 3.1: Plot of a simulation of the simple HelloWorld model.

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 initialised 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 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 initialised 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, without re-compilation if a is changed through the environment's parameter update facility.

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 pseudo-function der( ).

Our second example is only slightly more complicated, containing five rather simple equations:

$$m\dot{v}_x = -\frac{x}{L}F \tag{3.2}$$

$$m\dot{v}_y = -\frac{y}{L}F - mg \tag{3.3}$$

$$\dot{x} = v_x \tag{3.4}$$

$$\dot{y} = v_y \tag{3.5}$$

$$x^2 + y^2 = L^2 \tag{3.6}$$

This example is actually a mathematical model of a physical system, a planar pendulum, as depicted in Figure 3.2.

[3] plot is the OpenModelica command for plotting simulation results. The corresponding Dymola command would be plot({"x"}) respectively.

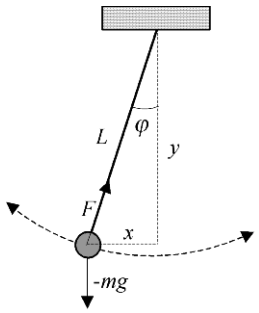

Fig. 3.2: A planar pendulum.

$$v_x \tag{3.7}$$

$$x^2 + y^2 = L^2 \tag{3.8}$$

$$v_y \tag{3.9}$$

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. 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;
```

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).

We simulate the Pendulum model and plot the x-coordinate, shown in Figure 3.3:

```
simulate(Pendulum, stopTime=4)

plot(x);
```

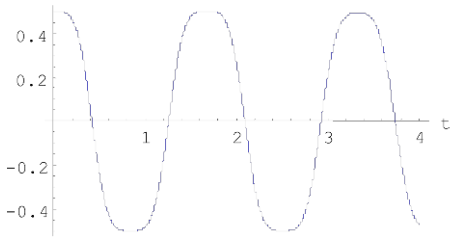

Fig. 3.3: Plot of a simulation of the Pendulum DAE model.

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 a section further below.

3.2.1 Variables and Predefined Types

This example shows a slightly more complicated model, which describes a Van der Pol[4] oscillator. Notice that here the keyword model is used instead of class with almost the same meaning.

```
model VanDerPol "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
  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.

A *parameter* is a special kind of constant which is implemented as a static variable that is initialised 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 behaviour of a model, e.g., changing the parameter lambda which strongly influences the behaviour 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.

[4] 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 behaviour of nonlinear vacuum tube circuits

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” predefined data types to support floating-point, integer, boolean, and string values. These predefined primitive types contain data that Modelica understands directly, as opposed to class types defined by programmers. There is also the Complex type for complex numbers computations, which is predefined in a library. The type of each variable must be declared explicitly. The predefined basic data types of Modelica are:

Boolean	either true or false
Integer	32-bit two’s complement, usually corresponding to the C int data type
Real	64-bit floating-point usually corresponding to the C double data type
String	string of text characters
enumeration(...)	enumeration type of enumeration literals
Clock	clock type for clocked synchronous models
Complex	for complex number computations, a basic type predefined in a library

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 behaviour 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 3.4):

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

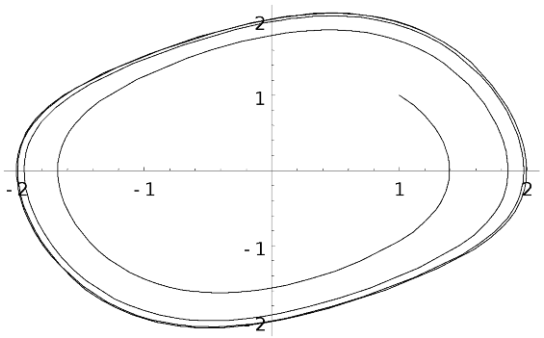

Fig. 3.4: Parametric plot of a simulation of the Van der Pol oscillator model.

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’.

3.2.2 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 the section about annotations at the end of this chapter.

3.2.3 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 (often called *parameters*) can usually be accessed and changed from Modelica tool graphical user interfaces. Most Modelica tools support re-simulating a model without re-compilation after changing a parameter, which usually makes the time waiting for results much shorter.

Parameters can be declared without a declaration equation (but a default start value is recommended) since their value can be defined, e.g., interactively or by reading from a file, before simulation starts. This is not possible for constants with prefix constant. For example:

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

A parameter with a default start value as shown below means that the parameter has a value, but that this value is expected to be replaced by a more realistic value, e.g., by the user, at a real simulation:

```
parameter Real mass(start=0), gravity(start=0), length(start=0);
```

Constants and parameters are very similar, but also have some differences. Parameters are typically much more common in application models. When in doubt whether to use constant or parameter, use parameter, except when declaring a constant in a package/library where constant is allowed but not parameter.

3.2.4 Variability

We have seen that some variables can change value at any point in time whereas named constants are more or less constant. In fact, there is a general concept of four levels of variability of variables and expressions in Modelica:

- Expressions or variables with *continuous-time variability* can change at any point in time.
- *Discrete-time variability* means that value changes can occur only at so-called events. There are two kinds: *unclocked discrete-time variability* and *clocked discrete-time variability*. In the latter case the value can only change at clock tick events.
- *Parameter variability* means that the value can be changed at initialisation before simulation, but is fixed during simulation.
- *Constant variability* means the value is always fixed. However, a named constant definition equation can be replaced by a so-called redeclaration or modification.

3.2.5 Default Start Values

If a numeric variable lacks a specified definition value or start value in its declaration, it is usually initialised 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*.

3.3 Object-Oriented Mathematical Modelling

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 emphasises sending messages between (dynamically) created objects.

The Modelica view on object-orientation is different since the Modelica language emphasises *structured* mathematical modelling. 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 modelling, can be summarised as follows:

- Object-orientation is primarily used as a *structuring* concept, emphasising 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*[5].
- An object is a collection of *instance* variables and equations that share a set of data.

However:

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

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

The declarative object-oriented way of describing systems and their behaviour 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 behaviour 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.

3.3.1 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 modelling and programming problems. Every object in Modelica has a class that defines its data and behaviour. 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 behaviour 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 behaviour 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 writingmyPoint.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.

3.3.1.1 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 when allocating on the heap. 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 classTriangle 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.

3.3.1.2 Initialization

Another problem is initialisation of variables. As mentioned previously in the section about variability, 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=Point(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=Point(1,2,3));
  Foo1 f1;
end Main;
```

A more general way of initialising 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 initialised in steady-state, i.e., when derivatives should be zero, is given below:

```
model Controller
parameter
  Real a=1, b=2;
  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;
```

3.3.1.3 Specialised 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, type, package, and function can be used to denote a class under appropriate conditions, called restrictions. Some of the specialised classes also have additional capabilities, called enhancements. For example, a function class has the enhancement that it can be called, whereas a record is a class used to declare a record data structure and has the restrictions that it may not contain equations and may only contain public declarations.

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

A model is almost the same as a class, i.e., those keywords are completely interchangeable. A block is a class with fixed causality, which means that for each connector variable of the class it is specified whether it has input or output causality. Thus, each connector 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, may not contain equations, and has only public sections, but has the additional property to allow connect(...) to instances of connector classes. A type is a class that can be an alias or an extension to a predefined type, enumeration, or array of a type. For example:

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

The idea of specialised classes has some advantages since they re-use most properties of the general *class concept*. The notion of specialised classes also 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 of specialised class 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 specialised classes. Furthermore, the construction of Modelica translators is simplified since only the syntax and semantics of the general class concept has to be implemented along with some additional checks on restrictions and enhancements of the classes.

The package and function specialised classes in Modelica have much in common with the class concept but also have many additional properties, so called enhancements. Especially functions have quite a lot of enhancements, e.g., they can be called with an argument list, instantiated at run-time, etc. An operator class is similar to a package which is like a container for other definitions, but may only contain declarations of functions and is intended for user-defined overloaded operators.

3.3.1.4 Reuse of Classes by Modifications

The class concept is the key to reuse of modelling 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 a built-in predefined variable 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;
```

3.3.1.5 Built-in Classes

The built-in predefined type classes of Modelica correspond to the types Real, Integer, Boolean, String, enumeration(...), and Clock, 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);
```

3.3.2 Inheritance

One of the major benefits of object-orientation is the ability to extend the behaviour and properties of an existing class. The original class, known as the *superclass* or *base class*, is extended to create a more specialised version of that class, known as the *subclass* or *derived class*. In this process, the behaviour 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;
```

3.3.3 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 modelling and programming in addition to the usual class functionality.

There are essentially two cases of generic class parameterisation 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.

3.3.3.1 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 classC, 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 3.5).

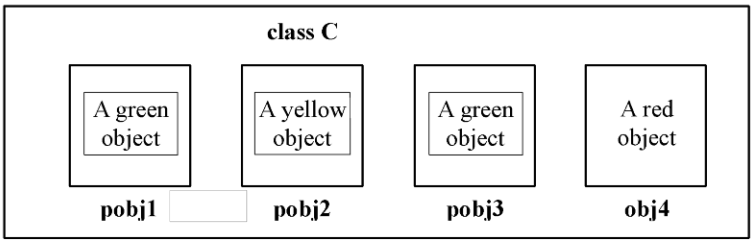

Fig. 3.5: Three class parameters pobj1, pobj2, and pobj3 that are instances (variables) of class C. These are essentially slots that can contain objects of different colors.

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;
```

3.3.3.2 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 3.6 depicts how the type value of the ColoredClass class parameter is propagated to the member object declarations obj1 and obj3.

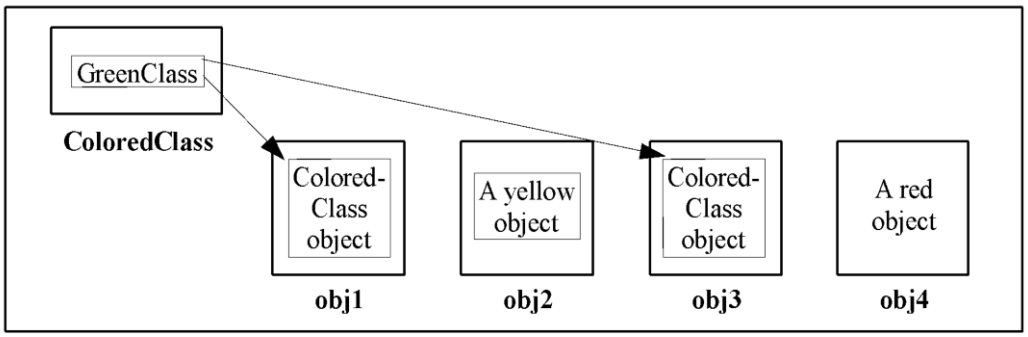

Fig. 3.6: The class parameter ColoredClass is a type parameter that is propagated to the two member instance declarations of obj1 and obj3.

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;
```

3.4 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 modelling capabilities and increased reuse potential of Modelica classes.

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

1. Assignment statements in conventional languages are usually represented as equations in Modelica.
2. Attribute assignments are represented as equations.
3. 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:

$$R \times i = v \tag{3.10}$$

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:

1. *Normal equations* occurring in equation sections, including the connect-equation, which is a special form of equation.
2. *Declaration equations*, which are part of variable or constant declarations.
3. *Modification equations*, which are commonly used to modify attributes.
4. *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 initialiser in other languages since it always holds and can be solved together with other equations in the total system of equations.

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);
```

3.4.1 Repetitive Equation Structures

Before reading this section you might want to take a look at the section further down about arrays, and the section further down about statements and algorithmic for-loops.

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. 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:

$$y = a[1] + a[2] \times x + a[3] \times x^2 + ... + a[n+1] \times x^n \tag{3.11}$$

The polynomial equation Eq. 3.11 above 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[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:

```
powers[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 $n + 1$.

3.4.2 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 yet part of the official Modelica language.

3.5 Acausal Physical modelling

Acausal modelling is a declarative modelling style, meaning modelling 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 modelling*. The term *physical modelling* reflects the fact that *acausal modelling* is very well suited for representing the *physical structure* of modelled systems.

The main advantage with acausal modelling 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.

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

To illustrate the idea of acausal physical modelling we give an example of a simple electrical circuit (Figure 3.7). The connection diagram[6] 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 modelling* is quite appropriate.

[6] A connection diagram emphasises 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.

The Modelica SimpleCircuit model below directly corresponds to the circuit depicted in the connection diagram of Figure 3.7. 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 later in this chapter.

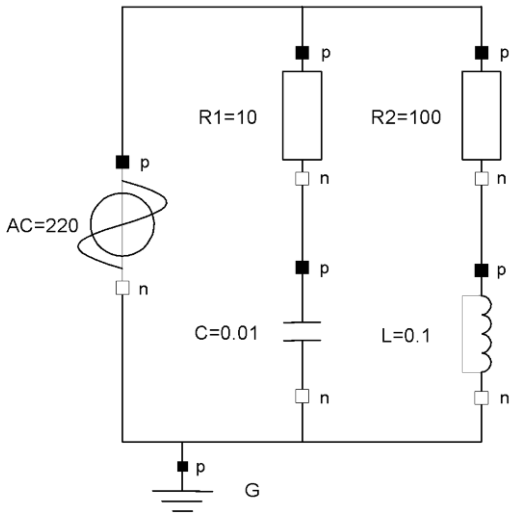

Fig. 3.7: Connection diagram of the acausal SimpleCircuit model.

```
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);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end SimpleCircuit;
```

As a comparison, we show the same circuit modelled using causal block-oriented modelling depicted as a diagram in Figure 3.8. Here the physical topology is lost–the structure of the diagram has no simple correspondence to the structure of the physical circuit board.

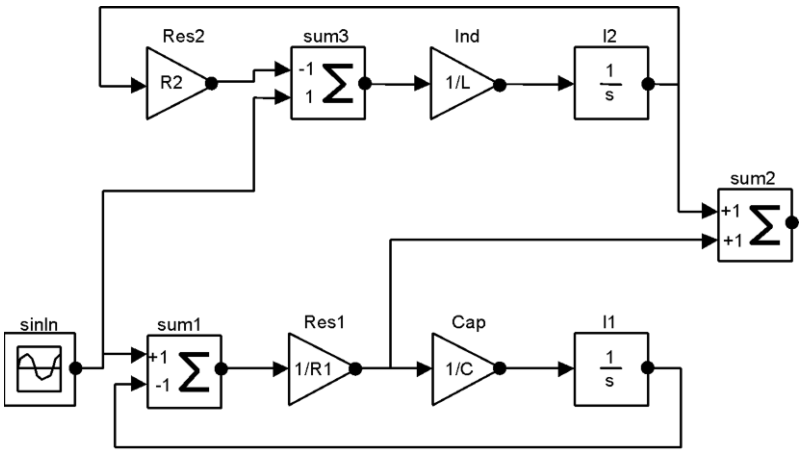

Fig. 3.8: The simple circuit model using causal block-oriented modelling with explicit signal flow.

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.

3.6 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 standardisation 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 visualised 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.

3.6.1 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 modelled 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 modelling.

3.6.2 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 3.9. We have earlier presented a connection diagram of a simple circuit in Figure 3.7.

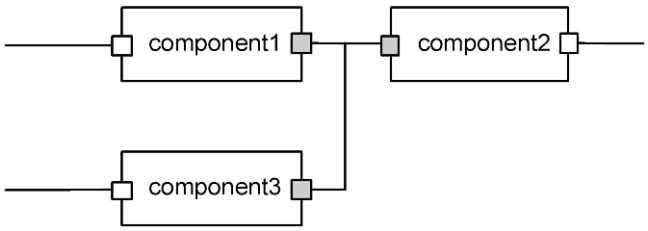

Fig. 3.9: Schematic picture of a connection diagram for components.

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 realised 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.

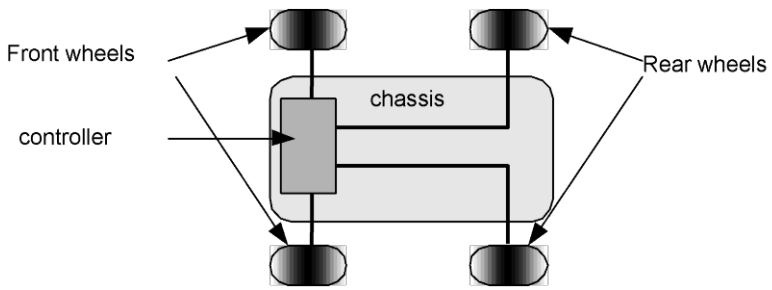

Fig. 3.10: A connection diagram for a simple car model.

A simple car example of a connection diagram for an application in the mechanical domain is shown in Figure 3.10.

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;
```

3.6.3 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 3.11) 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");
```

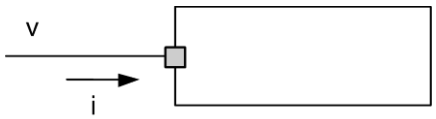

Fig. 3.11: A component with one electrical Pin connector.

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;
```

3.6.3.1 Connections

Connections between components can be established between connectors of equivalent type. Modelica supports equation-based acausal connections, which means that connections are realised 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 potential (default), or declared using the flow prefix:

1. Equality coupling, for potential 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.

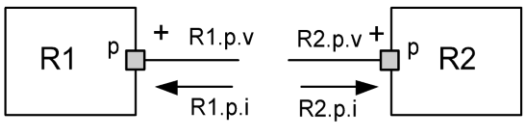

Fig. 3.12: Connecting two components that have electrical pins.

Connection equations are used to connect instances of connection classes. A connection equation connect(R1.p,R2.p), with R1.p and R2.p of connector class Pin, connects the two pins (Figure 3.12) so that they form one node. This produces two equations, namely:

```
R1.p.v = R2.p.v;
R1.p.i + R2.p.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.

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.

3.6.3.2 Implicit Connections with inner/outer

So far we have focused on explicit connections between connectors where each connection is explicitly represented by a connect-equation and a corresponding line in a connection diagram. However, when modelling certain kinds of large models with many interacting components this approach becomes rather clumsy because of the large number of potential connections–a connection might be needed between each pair of components. This is especially true for system models involving *force fields*, which lead to a maximum of $n \times n$ connections between the n components influenced by the force field or $1 \times n$ connections between a central object and n components if inter-component interaction is neglected.

For the case of $1 \times n$ connections, instead of using a large number of explicit connections, Modelica provides a convenient mechanism for *implicit connections* between an object and n of its components through the inner and outer declaration prefixes.

A rather common kind of implicit interaction is where a *shared attribute* of a single environment object is *accessed* by a number of components within that environment. For example, we might have an environment incl uding house components, each accessing a shared environment temperature, or a circuit board environment with electronic components accessing the board temperature.

A Modelica environment-component example model along these lines is shown below, where a shared environment temperature variable T0 is declared as a *definition declaration* marked by the keyword inner. This declaration is implicitly accessed by the *reference declarations* of T0 marked by the prefix outer in the components comp1 and comp2.

```
model Environment
  inner
    Real T0; //Definition of actual environment temperature T0
    Component comp1, comp2; // Lookup match comp1.T0 = comp2.T0 = T0
  parameter
    Real k=1;
  equation
    T0 = sin(k*time);
end Environment;
```

```
model Component
  outer
     Real T0; // Reference to temperature T0 defined in the environments
        Real T;
  equation
     T = T0;
end Component;
```

3.6.3.3 Expandable Connectors for Information Buses

It is common in engineering to have so-called information buses with the purpose to transport information between various system components, e.g. sensors, actuators, control units. Some buses are even standardised (e.g., IEEE), but usually rather generic to allow many kinds of different components.

This is the key idea behind the expandable connector construct in Modelica. An expandable connector acts like an information bus since it is intended for connection to many kinds of components. To make this possible it automatically expands the expandable connector type to accommodate all the components connected to it with their different interfaces. If an element with a certain name and type is not present, it is added.

All fields in an expandable connector are seen as connector instances even if they are not declared as such, i.e., it is possible to connect to e.g. a Real variable.

Moreover, when two expandable connectors are connected, each is augmented with the variables that are only declared in the other expandable connector. This is repeated until all connected expandable connector instances have matching variables, i.e., each of the connector instances is expanded to be the union of all connector variables. If a variable appears as an input in one expandable connector, it should appear as a non-input in at least one other expandable connector instance in the connected set. The following is a small example:

```
expandable connector EngineBus
end EngineBus;

block Sensor
  RealOutput speed;
end Sensor;

block Actuator
  RealInput speed;
end Actuator;

model Engine
   EngineBus bus;
   Sensor sensor;
   Actuator actuator;
equation
    connect(bus.speed, sensor.speed); // provides the non-input
    connect(bus.speed, actuator.speed);
end Engine;
```

3.6.3.4 Stream Connectors

In thermodynamics with fluid applications where there can be bi-directional flows of matter with associated quantities, it turns out that the two basic variable types in a connector–potential variables and flow variables–are not sufficient to describe models that result in a numerically sound solution approach. Such applications typically have bi-directional flow of matter with convective transport of specific quantities, such as specific enthalpy and chemical composition

If we would use conventional connectors with flow and potential variables, the corresponding models would include nonlinear systems of equations with Boolean unknowns for the flow directions and singularities around zero flow. Such equation systems cannot be solved reliably in general. The model formulations can be simplified when formulating two different balance equations for the two possible flow directions. This is however not possible only using flow and potential variables.

This fundamental problem is addressed in Modelica by introducing a third type of connector variable, called *stream variable*, declared with the prefix stream. A stream variable describes a quantity that is carried by a flow variable, i.e., a purely convective transport phenomenon.

If at least one variable in a connector has the stream prefix, the connector is called a *stream connector*and the corresponding variable is called a *stream variable*. For example:

```
connector FluidPort ...
  flow Real m_flow "Flow of matter; m_flow>0 if flow into component";
  stream Real h_outflow "Specific variable in component if m_flow < 0"
end FluidPort

model FluidSystem ...
   FluidComponent m1, m2, ..., mN;
   FluidPort c1, c2, ..., cM;
equation connect(m1.c, m2.c); ...
    connect(m1.c, cM); ...
end FluidSystem;
```

3.6.4 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 behaviour, and is therefore prefixed by the keyword partial. Partial classes are usually known as *abstract classes* in other object-oriented languages. Since a partial class is incomplete it cannot be used as a class to instantiate a data object.

```
partial class TwoPin8 "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;
```

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 modelling 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.

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 (Figure 3.13). 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.

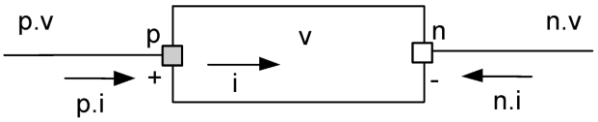

Fig. 3.13: Generic TwoPin class that describes the general structure of simple electrical components with two pins.

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.

3.6.5 Reuse of Partial Classes

Given the generic partial class TwoPin, it is now straightforward to create the more specialised 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 3.14).

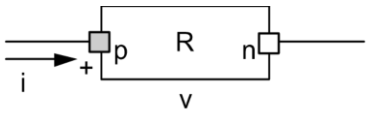

Fig. 3.14: A resistor component.

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

A class for electrical capacitors can also reuse TwoPin in a similar way, adding the constitutive equation for a capacitor (Figure 3.15).

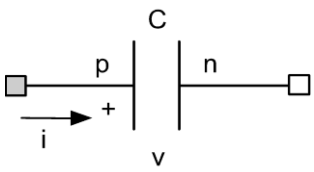

Fig. 3.15: A capacitor component.

```
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 $C.\dot{v}(t) = i(t)$ for all values of t, fulfilling the constitutive equation for the capacitor.

3.7 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 modelling of complex systems.

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.

Resistor

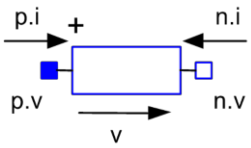

Fig. 3.16: Resistor component.

Four equations can be extracted from the resistor model depicted in Figure 3.14 and Figure 3.16. 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
```

Capacitor

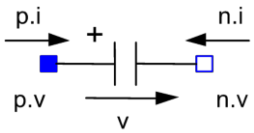

Fig. 3.17: Capacitor component.

The following four equations originate from the capacitor model depicted in Figure 3.15 and Figure 3.17, 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);
```

Inductor

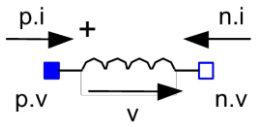

Fig. 3.18: Inductor component

The inductor class depicted in Figure 3.18 and shown below gives a model for ideal electrical inductors.

```
class Inductor "Ideal electrical inductor"
  extensd 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);
```

Voltage Source

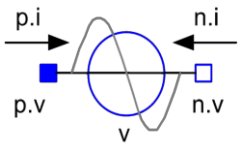

Fig. 3.19: Voltage source component VsourceAC, where $v(t) = VA \times sin(2 \times PI \times f \times time)$.

A classVsourceAC for the sin-wave voltage source to be used in our circuit example is depicted in Figure 3.19 and can be defined as below. This model as well as other Modelica models specify behaviour that evolves as a function of time. Note that the built-in 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);
```

Ground

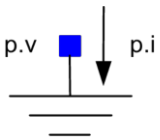

Fig. 3.20: Ground component.

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 3.20).

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

A single equation can be extracted from the Ground class.

```
  p.v = 0
```

3.7.1 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 3.21.

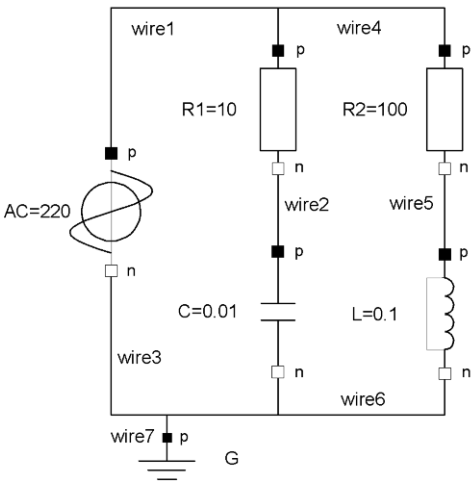

Fig. 3.21: The SimpleCircuit model.

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);  // Wire1, Capacitor circuit
  connect(R1.n, C.p);   //    Wire 2
  connect(C.n, AC.n);   //    Wire 3
  connect(R1.p, R2.p);  // Wire4, Inductor circuit
  connect(R2.n, L.p);   //    Wire 5
  connect(L.n, C.n);    //    Wire 6
  connect(AC.n, G.p);   // Wire7, Ground
end SimpleCircuit;
```

3.7.2 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, or by Boolean indexes from false to true, or by enumeration indexes. 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 kth dimension. Submatrices can be formed by utilising 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 greater than 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 $n_1 \times n_2 \times n_3 \times ...$ zero-filled integer array.
ones(n1,n2,n3,...)	Returns an $n_1 \times n_2 \times n_3 \times ...$ one-filled integer array.
fill(s,n1,n2,n3, ...)	Returns the $n_1 \times n_2 \times n_3 \times ...$ 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 generalised 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 kth dimension. For example, cat(1, {2,3}, {5,8,4}) gives the result {2,3,5,8,4}.

$$m \times n$$

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 matrix:

[$expr_{11}$, $expr_{12}$, ... $expr_{1n}$;

$expr_{21}$, $expr_{22}$, ... $expr_{2n}$;

. . .

$expr_{m1}$, $expr_{m2}$, ... $expr_{mn}$]

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}}]

$$2 \times 2$$

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

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

$$1 \times 1$$

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

```
[1]
```

gives the matrix:

```
{{1}}
```

3.8 Algorithmic Constructs

Even though equations are eminently suitable for modelling physical systems and for a number of other tasks, there are situations where non-declarative 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.

3.8.1 Algorithm Sections and Assignment Statements

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, end, or annotation.

```
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.

3.8.2 Statements

In addition to assignment statements, which were used in the previous example, a few other kinds of "algorithmic" statements are available in Modelica: if-then-else statements, for-loops, while-loops, return statements, etc. 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 previously presented in the section on repetitive equation structures:

$$y := a[1] + a[2] * x + a[3] * x\text{^}1 + ... + a[n + 1] * x\text{^}n;$$

When using equations to model the computation of the polynomial it was necessary to introduce an auxiliary 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;
```

3.8.3 Functions

Functions are a natural part of any mathematical model. A number of mathematical functions like abs, sqrt, mod,sin, cos, exp, etc. are both predefined in the Modelica language and 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 (except impure Modelica functions marked with the keyword impure). 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 formal 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.

3.8.4 Operator Overloading and Complex Numbers

Function and operator overloading allow 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.

In fact, overloading already exists predefined to a limited extent for certain operators in the Modelica language. For example, the plus (+) operator for addition has several different definitions depending on the data type:

1. 1+2 — means integer addition of two Integer constants giving an integer result, here 3.
2. 1.0+2.0 — means floating point number addition of two Real constants giving a floating-point number result, here 3.0.
3. "ab"+"2" — means string concatenation of two string constants giving a string result, here "ab2".
4. {1,2} +{3,4} — means integer vector addition of two integer constant vectors giving a vector result, here {4,6}.

Overloaded operators for user-defined data types can be defined using operator record and operator function declarations. Here we show part of a complex numbers data type example:

```
operator record Complex "Record defining a Complex number"

  Real re "Real part of complex number";
  Real im "Imaginary part of complex number";

  encapsulated operator 'constructor'
    import Complex;
    function fromReal
      input Real re;
      output Complex result = Complex(re=re, im=0.0);
        annotation(Inline=true);
    end fromReal;
  end 'constructor';

  encapsulated operator function '+'
    import Complex;
    input Complex c1;
    input Complex c2;
    output Complex result "Same as: c1 + c2";
      annotation(Inline=true);
    algorithm
      result := Complex(c1.re + c2.re, c1.im + c2.im);
    end '+';
  end Complex;
```

In the above example, we start as usual with the real and imaginary part declarations of the re and im fields of the Complex operator record definition. Then comes a *constructor definition* fromReal with only one input argument instead of the two inputs of the default Complex constructor implicitly defined by the Complex record definition, followed by overloaded operator definition for **'+'**.

How can these definitions be used? Take a look at the following small example:

```
Real    a;
Complex b;
Complex c = a + b;   // Addition of Real number a and Complex number b
```

The interesting part is in the third line, which contains an addition a+b of a Real number a and a Complex number b. There is no built-in addition operator for complex numbers, but we have the above overloaded operator

definition of '+' for two complex numbers. An addition of two complex numbers would match this definition right away in the lookup process.

However, in this case we have an addition of a real number and a complex number. Fortunately, the lookup process for overloaded binary operators can also handle this case if there is a constructor function in the Complex record definition that can convert a real number to a complex number. Here we have such a constructor called fromReal.

Note that the Complex is already predefined in a Modelica record type called Complex. This is distributed as part of the Modelica standard library (MSL) but defined at the top-level outside of MSL. It is referred to as Complex or .Complex. and is automatically preloaded by most tools, enabling direct usage without the need for any import statements (except in encapsulated classes). There is a library ComplexMath with operations on complex numbers.

The following are some examples of using the predefined type Complex:

```
Real a = 2;
Complex j = Modelica.ComplexMath.j;
Complex b = 2 + 3*j;
Complex c = (2*b + a)/b;
Complex d = Modelica.ComplexMath.sin(c);
Complex v[3] = {b/2, c, 2*d};
```

3.8.5 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;
```

3.8.6 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;
```

3.9 Discrete Event and Hybrid modelling

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 behaviour, 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*. There are two kinds of discrete-time variables: *unclocked discrete-time variables* which keep their values constant between events and *clocked discrete-time variables* which are undefined between associated clock-tick events, see Figure 3.22. Examples of discrete-time variables are Real variables declared with the prefix discrete, clock variables and clocked

variables, or Integer, Boolean, and enumeration variables which are discrete-time by default and cannot be continuous-time.

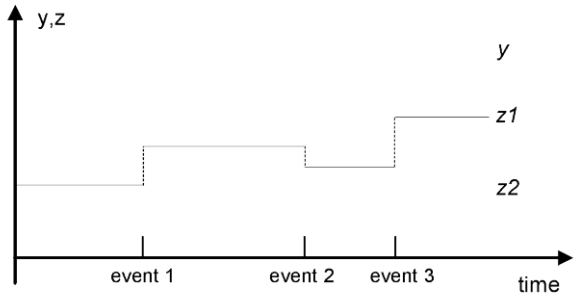

Fig. 3.22: Discrete-time variables *z1* and *z2* change value only at event instants, whereas continuous-time variables like *y* may change value both between and at events. An unclocked discrete-time variable *z1* is defined also between events, whereas a clocked discrete-time variable z2 is defined only at associated clock-tick 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 modelling techniques *hybrid modelling*. The introduction of hybrid mathematical models creates new difficulties for their solution, but the disadvantages are far outweighed by the advantages.

Modelica provides four kinds of constructs for expressing hybrid models: conditional expressions or equations to describe discontinuous and conditional models, when-equations to express equations that are valid only at discontinuities (e.g., when certain conditions become true), clocked synchronous constructs, and clocked state machines. For example, if-then-else conditional expressions allow modelling 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 3.23, and the ideal diode characteristic in parameterized form is shown in Figure 3.24.

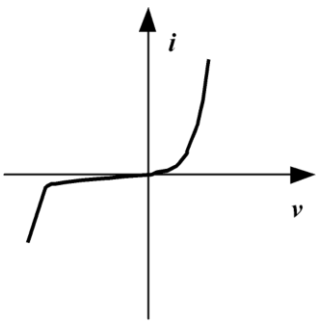

Fig. 3.23: Real diode characteristic.

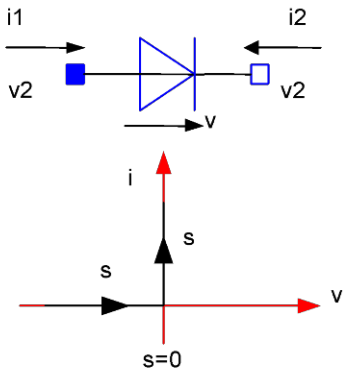

Fig. 3.24: Ideal diode characteristic.

Since the voltage level of the ideal diode would go to infinity in an ordinary voltage-current diagram, a parameterised 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;
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 modelled. The motion of the ball is characterised 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 3.25. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient of1 and would not lose any energy at a bounce. A more realistic ball, as the one modelled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

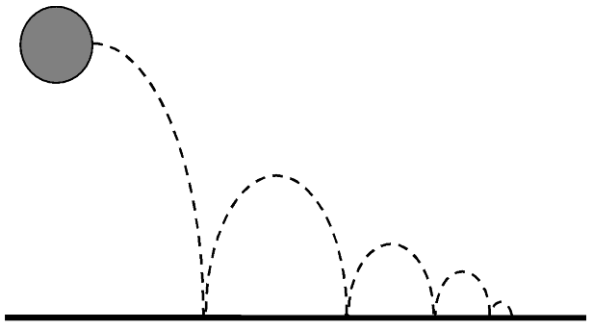

Fig. 3.25: Trajectory of bouncing ball.

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 reinitialisation: reinit(velocity,-c*pre(velocity)), which in this case reinitialises 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 behaviour of the simulation, called shattering, or the Zeno effect 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.

Synchronous Clocks and State Machines

Starting from Modelica version 3.3 two main groups of discrete-time and hybrid modelling features are available in the language: built-in synchronous clocks with constructs for clock-based modelling and synchronous clock-based state-machines.

As depicted in Figure 3.22, clocks and clocked discrete-time variable values are only defined at clock tick events for an associated clock and are undefined between clock ticks.

Synchronous clock-based modelling gives additional advantages for real-time and embedded system modelling in terms of higher performance and avoidance and/or detection of certain kinds of errors.

To give a very simple example, in the model below the clock variable clk is defined with an interval of 0.2 seconds. This clock is used for sampling the built-in time variable, with result put in x and defined only at clock ticks. Finally, the variable y is defined also between clock ticks by the use of the hold operator.

```
model BasicClockedSynchronous
  Clock clk "A clock variable";
  discrete Real x "Clocked discrete-time variable";
  Real y "Unclocked discrete-time variable";
equation
  clk = Clock(0.2); // Clock with an interval of 0.2 seconds
  x = sample(time,clk); // Sample the continuous time at clock ticks
  y = hold(x)+1; // Hold value is defined also between clock ticks
end BasicClockedSynchronous;
```

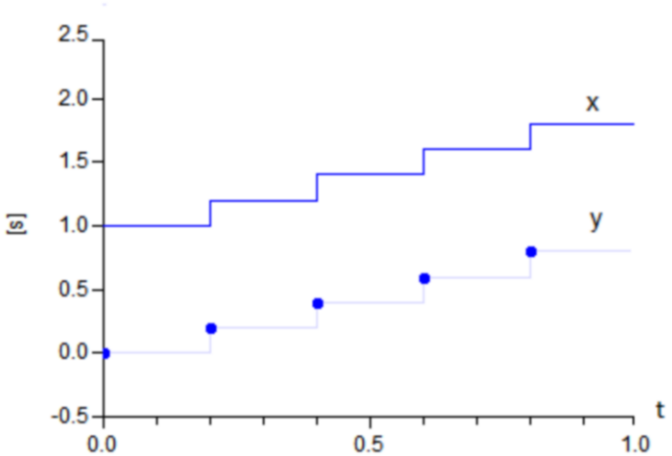

Fig. 3.26: Simulation of model BasicClockedSynchronous showing clocked discrete-time variable x and unclocked discrete-time variable y.

3.10 Modularity Facilites

3.10.1 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 most definitions in the Qt GUI library are prefixed with Q. 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 modelling problems since inheritance for import is not really a package specialisation. 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 nameModelica.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 MyTypes.*;
  import Modelica.SIunits.*;

  class Foo;
    Voltage v;
    Modelica.SIunits.Ampere i;
  end Foo;
end MyPack;
```

Importing definitions from one package into another class 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 type definition named Voltage is introduced into the package MyTypes, a compilation error would arise in the package MyPack since it is also defined in the package Modelica.SIunits.

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:

1. Visually scan the whole source code of the current package, which might be large.
2. 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 (unfortunately with one exception, names starting with a dot **.**, 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;
```

3.10.2 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(preserveAspectRatio = true,
  extent = {{-100,-100},{100,100}}, grid = {2,2}),
  graphics = {
    Rectangle(extent = {{-70,30},{70,-30}}, lineColor = {0,0,255},
    fillColor = {255,255,255},
      fillPattern = FillPattern.Solid),
    Line(points = {{-90,0},{-70,0}}, color = {0,0,255}),
    Line(points = {{70,0},{90,0}}, color ={0,0,255}),
    Text(extent = {{-144,-40},{142,-72}}, lineColor = {0,0,0},
    textString = "R=\%R"),
  ...
  ));
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 annotation Documentation is used for model documentation, as in this example for the Capacitor model:

```
    annotation(
      Documentation(info = "<HTML>
        <p>
         The linear capacitor connects the branch voltage <i>v</i> with
            the
         branch current <i>i</i> by <i>i = C * dv/dt</i>.
         The Capacitance <i>C</i> is allowed to be positive, zero, or
            negative.
        </p>
       </HTML>
     )
```

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");
```

3.10.3 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 capitalised, 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, often to characterise 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.

For more details on naming conventions see the documentation inside the package.

3.11 Modelica Standard Library

Much of the power of modelling 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 standardised 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*, abbreviated MSL. It provides constants, types, connector classes, partial models, and model classes of components from various application areas, which are grouped into sub-packages 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 under an open source license for both noncommercial and commercial purposes. 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 3.27 is one of the simplest examples illustrating this capability.

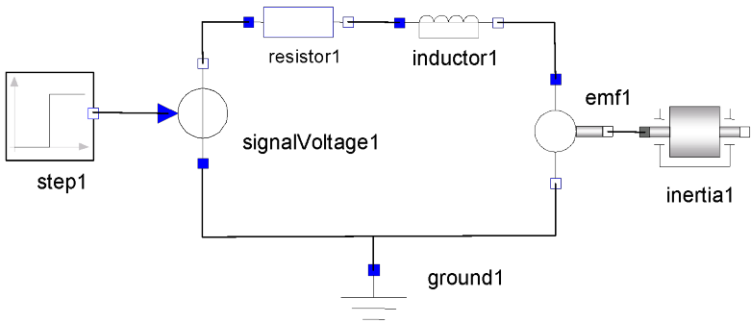

Fig. 3.27: A multidomain DCMotorCircuit model with mechanical, electrical, and signal block components.

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 3.28, where the DC-motor model is being constructed. The left window shows theModelica.Mechanics.Rotational library, from which icons can be dragged and dropped into the central window when performing graphic design of the model.

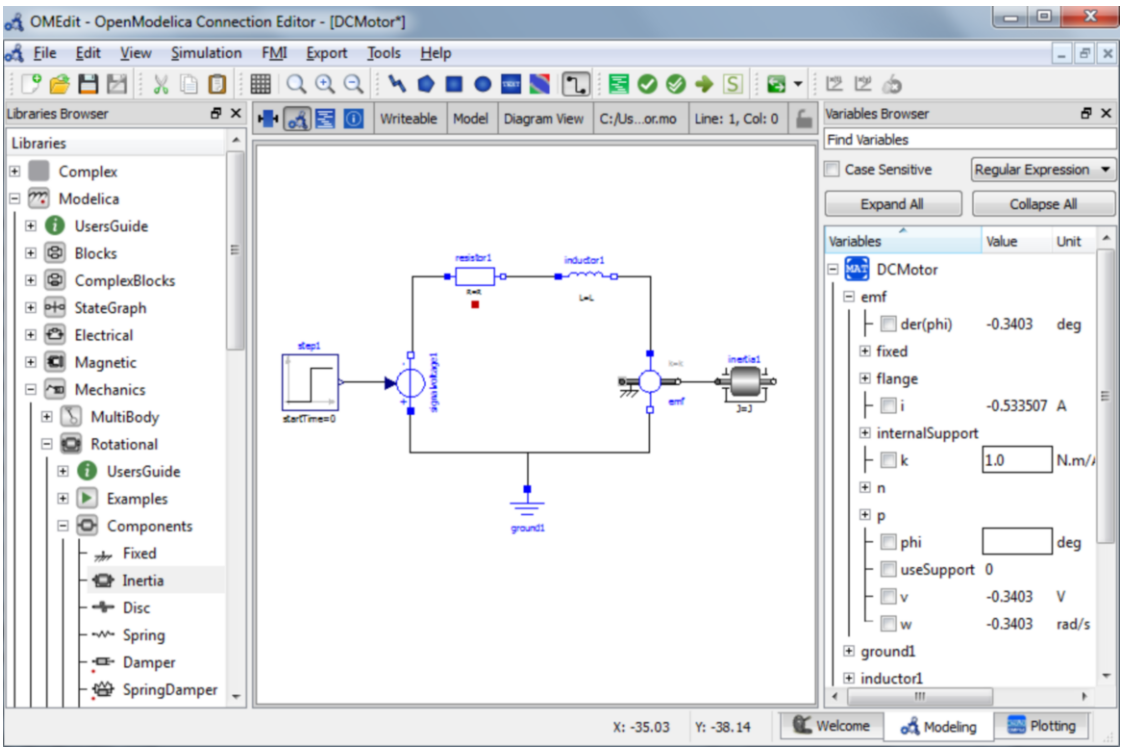

Fig. 3.28: Graphical editing of an electrical DC-motor model, with part of the model component hierarchy at the right and icons of the Modelica standard library in the left window.

3.12 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 typical process of translation and execution of a Modelica model, which is sketched in Figure 3.29. Note that this is the typical process used by most Modelica tools. There are alternative ways of executing Modelica, some of which

are currently at the research stage, that often keep more of the object-oriented structure at the later stages of translation and execution.

First the Modelica source code is parsed and converted into an internal representation, usually an abstract syntax tree. This representation is analysed, 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 optimiser module containing algebraic simplification algorithms, symbolic index reduction methods, etc., eliminates most equations, keeping only a minimal set that eventually will be solved numerically, see also the chapter in this book about numerical approximation methods.

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.

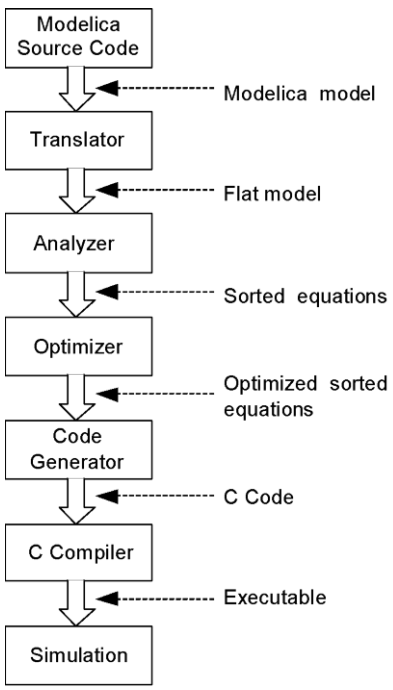

Fig. 3.29: The typical stages of translating and executing a Modelica model.

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.

3.12.1 Hand Translation of the Simple Circuit Model

Let us return once more to the simple circuit model, previously depicted in Figure 3.7, but for the reader's convenience also shown below in Figure 3.30. It is instructive to translate this model by hand, in order to understand the process.

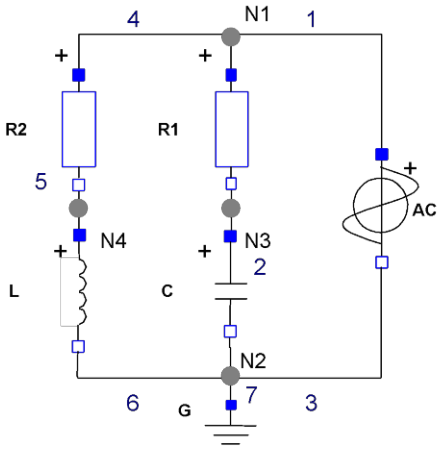

Fig. 3.30: The SimpleCircuit model once more, with explicitly labeled connection nodes N1, N2, N3, N4, and wires 1 to 7.

Classes, instances and equations are translated into a flat set of equations, constants, and variables (see the equations in Table 3.1), 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 potential 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 *potential* variable representing voltage potential. Therefore, the equality equation R1.p.v=AC.p.v is generated. Equality equations are always generated when potential 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 3.1) generated from the SimpleCircuit class consists of 32 differential-algebraic equations. These include 32 variables, as well as time and several parameters and constants.

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 3.1: The equations extracted from the simple circuit model–an implicit DAE system.

Table 3.2 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.

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			

Table 3.2: The variables extracted from the simple circuit model.

3.12.2 Transformation to State Space Form

The implicit differential algebraic system of equations (DAE system) in Table 3.1 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.
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 an explicit state space form as below.

$$\dot{x} = f(x,t) \tag{3.12}$$

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:

$$\dot{x} = \{C.v, L.i\}, u = \{AC.v\} \quad \text{(with constants: R1.R, R2.R, C.C, L.L, AC.VA, AC.f, AC.PI)} \tag{3.13}$$

$$\dot{x} = \{\mathbf{der}(C.v), \mathbf{der}(L.i)\} \tag{3.14}$$

3.12.3 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. $\mathbf{der}(x)$) at time t, e.g.:

$$\mathbf{der}(x) = (x_{t+h} - x_t)/h \tag{3.15}$$

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

$$x_{t+h} = x_t + \mathbf{der}(x) \times h \tag{3.16}$$

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:

$$\mathbf{der}(C.v) = C.i/C.C \tag{3.17}$$

$$\mathbf{der}(L.i) = L.v/L.L \tag{3.18}$$

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 (3.19) through (3.21) below.

$$C.i = R1.v/R1.R$$
$$\text{using: } C.i = C.p.i = -R1.n.i = R1.p.i = R1.i = R1.v/R1.R \tag{3.19}$$

$$R1.v = R1.p.v - R1.n.v = R1.p.v - C.v$$
$$\text{using: } R1.n.v = C.p.v = C.v + C.n.v = C.v + AC.n.v = C.v + G.p.v = C.v + 0 = C.v \tag{3.20}$$

$$R1.p.v = AC.p.v = AC.VA * sin(2 * AC.f * AC.PI * t)$$
$$\text{using: } AC.p.v = AC.v + AC.n.v = AC.v + G.p.v = AC.VA \times sin(2 \times AC.f \times AC.PI \times t) + 0 \quad (3.21)$$

In a similar fashion, we derive equations (3.22) and (3.23) below:

$$L.v = L.p.v - L.n.v = R1.p.v - R2.v$$
$$\text{using: } L.p.v = R2.n.v = R1.p.v - R2.v \text{ and } L.n.v = C.n.v = AC.n.v = G.p.v = 0 \quad (3.22)$$

$$R2.v = R2.R \times L.p.i$$
$$\text{using: } R2.v = R2.R \times R2.i = R2.R \times R2.p.i = R2.R \times (-R2.n.i) = R2.R \times L.p.i = R2.R \times L.i \quad (3.23)$$

Collecting the five equations together:

$$\begin{aligned} C.i &= R1.v/R1.R \\ R1.v &= R1.p.v - C.v \\ R1.p.v &= AC.VA \times sin(2 \times AC.f \times AC.PI \times t) \\ L.v &= R1.p.v - R2.v \\ R2.v &= R2.R \times L.i \end{aligned} \quad (3.24)$$

By sorting the equations Eqs. 3.24 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 partitioning*, i.e., conversion of the equation system coefficient matrix into block lower triangular form (Table 3.3).

	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

Table 3.3: TBlock 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 emphasised 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})
```

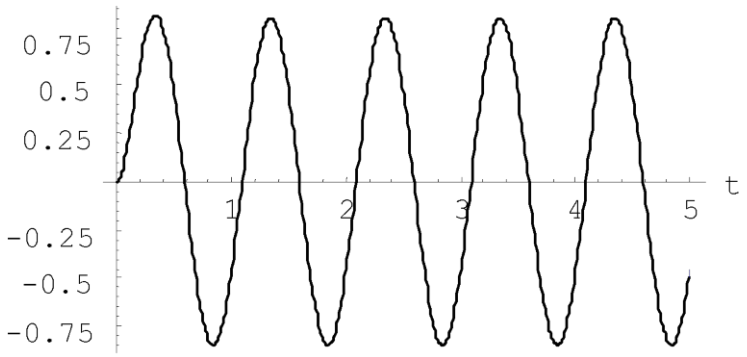

Fig. 3.31: Simulation of the SimpleCircuit model with a plot of the voltage C.v over the capacitor.

3.13 Tool Interoperability through Functional Mockup Interface

Even though Modelica is a universal modelling language that covers most application domains, there exist many established existing modelling and simulation tools in various application domains. There is a strong industrial need for interoperability and model interchange between tools. The Functional Mockup Interface standard (FMI, `www.fmi-standard.org`) has been developed to fulfil this need. It allows exporting models compiled into C-code combined with XML-based descriptors, which can be imported and simulated in other tools, both Modelica and non-Modelica tools. The standard also supports co-simulation between tools, and is already supported by a number of tool vendors.

3.14 Summary

This chapter has given a quick overview of the most important concepts and language constructs in Modelica. We have also defined important concepts such as object oriented mathematical modelling and acausal physical modelling, 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.

3.15 Literature and Further Reading

This chapter has given a short introduction to Modelica and equation-based modelling. A much more complete presentation of Modelica together with applications and the technology behind Modelica simulation tools can be found in [13].

The recent history of mathematical modelling languages is described in some detail in[14], whereas bits and pieces of the ancient human history of the invention and use of equations can be found in [122], and the picture on Newton's second law in latin in [144]. Early work on combined continuous/discrete simulation is described in [233] followed by [65]. A very comprehensive overview of numeric and symbolic methods for simulation is available in [68]. This author's first simulation work involving solution of the Schrödinger equation for a particular case is described in [113].

Current versions of several Modelica tools are described at `www.modelica.org`, including OpenModelica, Wolfram SystemModeler (before 2012 called MathModelica), and Dymola meaning the Dynamic modelling Laboratory. Several of the predecessors of the Modelica language are described in the following publications including Dymola meaning the Dynamic modelling Language [88, 89], Omola [203, 9], ObjectMath [119, 299, 120], NMF [248], Smile [93], etc.

Speed-Up, the earliest equation-based simulation tool, is presented in [251], whereas Simula-67–the first object-oriented programming language–is described in [35]. The early CSSL language specification is described in [258] whereas the ACSL system is described in [209]. The Hibliz system for an hierarchical graphical approach to modelling is presented in [89]. Software component systems are presented in [12, 260].

The Simulink system for block oriented modelling is described in [201], whereas the MATLAB language and tool are described in [202].

The DrModelica electronic notebook with the examples and exercises from this book has been inspired by DrScheme [102] and DrJava [8], as well as by Mathematica [306], a related electronic book for teaching mathematics [80], and the MathModelica (in 2012 renamed to Wolfram SystemModeler) environment [117, 229, 110]. The first version of DrModelica is described in [185, 186].

The OpenModelica version of notebook called OMNotebook is described in [103, 11]. Applications and extensions of this notebook concept are presented in [250, 210, 271].

General Modelica articles and books: [204, 117, 90], a series of 17 articles (in German) of which [225](Otter 1999) is the first, [270, 114, 91, 107, 111, 112].

The proceedings from the following conferences, as well as several conferences not listed here, which contain a number of Modelica related papers: the International Modelica Conference: [105, 226, 106, 255, 172, 18, 64, 75] and the Equation-Based Object-Oriented Languages and Tools (EOOLT) workshop series: [116, 115, 118, 66, 217]. the Scandinavian Simulation Conference, e.g.: [104] and some later conferences in that series, two special conferences with focus on biomedical [109] and safety issues [108], respectively.

3.16 Self-assessment

1. Create a class `Add` that calculates the sum of two parameters, which are Integer numbers with given values.
2. Create an instance of class Dog that overrides the default name with `"Timmy"`

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

3. What do the terms **partial**, **class**, and **extends** stand for?
4. Consider the `Bicycle` class below.

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

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

5. 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. 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:[7]

```
val(martinsBirthday.birthYear,0)
```

6. Consider the class `Ptest` below:

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

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

7. Create the class `Average` that 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.
```

8. 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 as suggested in the question above.
9. Using an *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.
10. Using a *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`.

Acknowledgements

This work has been supported by Vinnova in the ITEA3 OPENCPS project and in the RTISIM project. Support from the Swedish Government has been received from the ELLIIT project, as well as from the European Union in the H2020 INTO-CPS project. The OpenModelica development is supported by the Open Source Modelica Consortium.

[7] In OpenModelica, the expression val(martinsBirthday.birthYear,0) means the birthYear value at time=0, at the beginning of the simulation. Interpolated values at other simulated time points can also be obtained. Constant expressions such as 33+5, and function calls, can also be evaluated interactively.