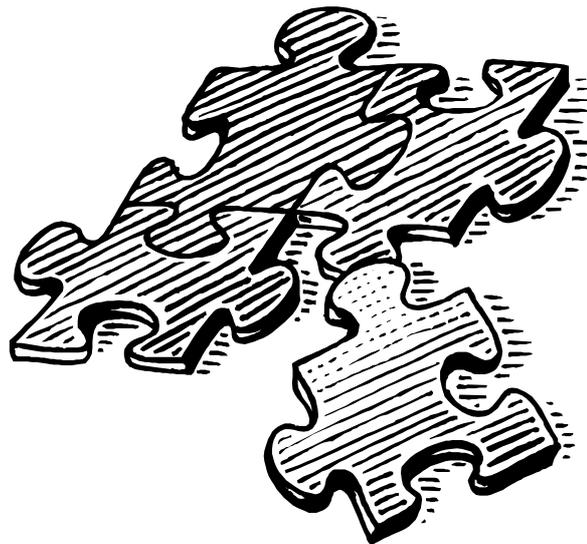


Peter Fritzson

Introduction to Modelica

the Language for
Physical Modeling and Simulation



– Incomplete Draft –

Table of Contents

1. A Quick Tour of Modelica	7
1.1 Getting started.....	7
1.1.1 Variables.....	9
1.1.2 Comments.....	10
1.1.3 Constants	11
1.2 Object-Oriented Mathematical Modeling.....	11
1.3 Classes and Instances.....	12
1.3.1 Creating Instances	13
1.3.2 Initialization.....	14
1.3.3 Restricted Classes.....	14
1.3.4 Reuse of Modified Classes.....	15
1.3.5 Built-in Classes	16
1.4 Inheritance.....	16
1.5 Generic Classes.....	17
1.5.1 Class Parameters being Components.....	17
1.5.2 Class Parameters being Types.....	18
1.6 Equations	18
1.6.1 Repetitive Equation Structures.....	20
1.7 Acausal Physical Modeling.....	20
1.7.1 Physical Modeling vs. Block Oriented Modeling.....	21
1.8 The Modelica Software Component Model.....	22
1.8.1 Components	23
1.8.2 Connection Diagrams	23
1.8.3 Connectors and Connector Classes.....	24
1.8.4 Connections	25
1.9 Partial Classes Describe Common Properties.....	26
1.9.1 Reuse of Partial Classes.....	26
1.10 Electrical Component Library	27
1.10.1 Resistor	27
1.10.2 Capacitor.....	28
1.10.3 Inductor.....	28
1.10.4 Voltage Source	29
1.10.5 Ground.....	29
1.11 The Simple Circuit Model	30
1.12 Arrays.....	31
1.13 Algorithmic Constructs	33
1.13.1 Algorithms	33
1.13.2 Statements.....	34
1.13.3 Functions.....	35
1.13.4 External functions.....	36

1.13.5	Algorithms Viewed as Functions.....	37
1.14	Hybrid Modeling	37
1.15	Packages	40
1.16	Implementation and Execution of Modelica	42
1.16.1	Hand Translation of the Simple Circuit Model.....	43
1.16.2	Transformation to State Space Form	45
1.16.3	Solution Method	46
1.16.4	Event Iteration and the Pre Function.....	49

List of Figures

Figure 1.1– Plot of a simple simulation.....	8
Figure 1.2 – Plot of the Van der Pol oscillator model	10
Figure 1.3 - Connection diagram of the acausal simple circuit model.....	21
Figure 1.4 - The simple circuit model using causal block oriented modeling with explicit signal flow.	22
Figure 1.5 - Schematic picture of a connection diagram for components.....	23
Figure 1.6 - A connection diagram for simple a car model.....	24
Figure 1.7 - A component with one electrical Pin connector.	25
Figure 1.8 - Connecting two components that have electrical pins.	25
Figure 1.9 - Generic TwoPin class that describes the general structure of simple electrical components with two pins.....	26
Figure 1.10 - A resistor component.....	26
Figure 1.11 - A capacitor component.	27
Figure 1.12 - Resistor component	27
Figure 1.13 - Capacitor component.....	28
Figure 1.14 - Inductor component.....	28
Figure 1.15 - Voltage source component $V_{sourceAC}$, where $u(t) = V_A \sin(2 \cdot \pi \cdot f \cdot time)$	29
Figure 1.16 - Ground component.	29
Figure 1.17 - The simple circuit model.	30
Figure 1.18 - Real diode characteristic.....	38
Figure 1.19 - Ideal diode characteristic.	39
Figure 1.20 - A bouncing ball.....	40
Figure 1.21 - The stages of translating and executing a Modelica model.....	43
Figure 1.22 - The SimpleCircuit model once more.....	44

1. A Quick Tour of Modelica

This is a short tour of the Modelica computer language that helps you getting started writing Modelica code quickly, as well as explaining the fundamentals of Modelica component-based modeling. Modelica is primarily a modeling language, sometimes called hardware description language, that allows you to specify mathematical models of complex physical systems, e.g. for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is also an object-oriented equation based programming language, oriented towards computational applications with high complexity requiring high performance. The four most important features of Modelica are:

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

This chapter gives a brief overview of the main points of the language, without slowing the reader with the full details. Subsequent chapters contain detailed discussions of specific Modelica features

1.1 Getting started

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

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

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

$$x' = -a * x$$

The variable x in this equation is a state variable that can change value over time. The derivative x' is the time derivative of x . Since all Modelica programs, usually called models, consist of class declarations, our Hello World 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 code¹. Then invoke the simulation command in your Modelica environment. This will compile the Modelica code to some intermediate code, usually C code, which in turn will be compiled to machine code and executed together with a numerical ordinary differential equation (ODE) solver or differential algebraic equation (DAE) solver to produce a solution for x as a function of time. The following command in the MathModelica environment produces a solution between time 0 and time 2:

```
Simulate [HelloWorld, {t, 0, 2}]
```

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

```
PlotSimulation[HelloWorld.x[t], {t, 0, 2}]
```

giving the curve below:

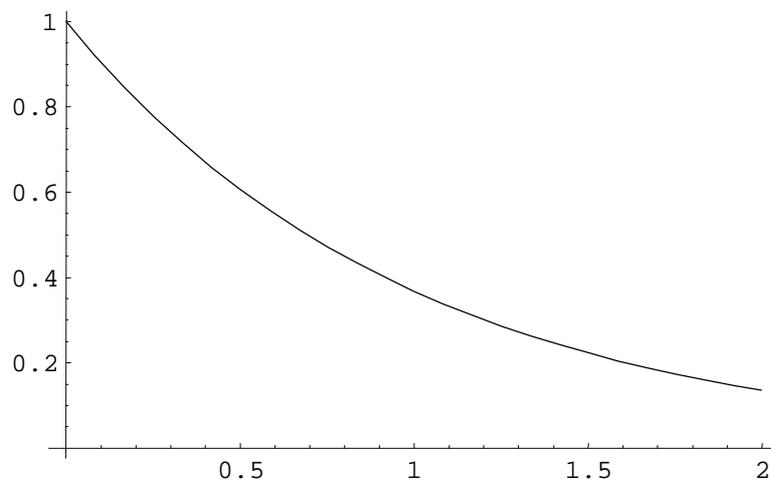


Figure 1.1– Plot of a simple simulation

Now you 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 fields and a single equation. The first field is the variable x which is initialized to a start value of 1 at the time when the simulation starts. All variables in Modelica have a start attribute with a default value which is normally set to 0. Having a different start value is accomplished by providing a so-called modifier within parentheses after the

¹ For example, there is one Modelica environment called MathModelica from MathCore, and another called Dymola from Dynasim.

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

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

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

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

1.1.1 Variables

The next example shows a slightly more complicated model, which describes a Van der Pol oscillator. The example also contains declarations of a few dummy variables to illustrate the primitive data types in Modelica. Notice that here the keyword `model` is used instead of `class` with 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) "y coordinate";           // y starts at 1
  parameter Real lambda = 0.3;
  Boolean bb;
  String dummy = "dummy string";
  Integer fooint = 0;
equation
  der(x) = y; // This is the first equation //
  der(y) = -x + lambda*(1 - x*x)*y; /* This is the second
                                     differential equation */
end VanDerPol;
```

This example contains declarations of two 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 simulation parameter.

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

Finally we have declarations of three dummy variables which are not really needed in the model of the oscillator: the `Boolean` variable `bb` which has a default initial value of `false` if nothing else is specified, the string variable `dummy`, and the integer variable `fooint`.

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

Boolean	either true or false
Integer	corresponding to the C <code>int</code> data type, usually 32-bit two's complement
Real	corresponding to the C <code>double</code> data type, usually 64-bit floating-point
String	string of 8-bit characters

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

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

```
Simulate2[VanDerPol, {t, 0, 25}]
```

A phase plane plot of the state variables for the Van der Pol oscillator model:

```
ParametricPlot[Evaluate[{x[t], y[t]}], {t, 0, 25}]
```

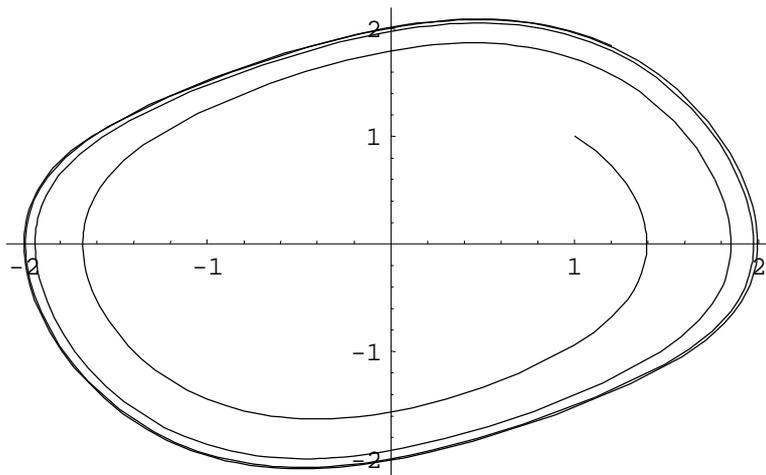


Figure 1.2 – Plot of the Van der Pol oscillator model

1.1.2 Comments

Arbitrary descriptive text, e.g. in English, scattered throughout a computer program, are comments to that code. Modelica has three styles of comments, all illustrated in the above example. Comments makes it possible to write descriptive text together with the code which makes it 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

² Simulate is the MathModelica notebook command for simulation.

that you find errors in your code when you write comments since when explaining your code you are forced to think about it.

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 can be present at Modelica class or variable definitions, and are processed to be used by the Modelica programming environment, e.g. to appear in menus or as help texts. 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 the Modelica developer. 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.

1.1.3 Constants

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

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

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

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

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

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

If a numeric variable lacks a specified definition value in its declaration, it is initialized to zero at the start of the simulation. Boolean variables are initialized to `false`, and string variables to the empty string if nothing else is specified. Exceptions to this rule are function results and local variables, where the default value is undefined.

1.2 Object-Oriented Mathematical Modeling

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

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

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

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

- ? Object-orientation is primarily used as a structuring concept, emphasizing the declarative structure and reuse of mathematical models.
- ? Dynamic model properties are expressed in a declarative way through equations.
- ? An object is a collection of instance variables and equations that share a set of stored data.
- ? Object-orientation is not viewed as dynamic message passing.

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

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

1.3 Classes and Instances

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

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

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

```
class Point "point in a three-dimensional space"
```

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

The `Point` class has three fields representing the `x`, `y`, and `z` coordinates of a point and has (as yet) no equations. A class declaration like this one is like a blueprint that defines how instances created from that class looks like, as well as instructions in the form of equations that define the behavior of those objects.

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 read or update 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.

1.3.1 Creating Instances

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

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

There is one remaining problem, however. In what context should `mypoints` 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 “last” class in a Modelica program as a kind of “main” class that is always instantiated, implying that its fields are instantiated, and that the fields of those fields are instantiated, etc. Therefore, to instantiate `mypoints`, either make the class `mypoints` the last class or declare an instance of `mypoints` in the last class. In the following example, both the class `mypoints` and the class `foo1` are instantiated. Strictly speaking, this “main” class actually does not need to be last since Modelica allows using a class before it is declared; it just needs to be at the top of the instantiation hierarchy in the Modelica program to be executed.

```
class foo1  
  ...  
end foo1;  
  
class foo2  
  ...  
end foo2;  
...  
  
class mypoints
```

```
Point point1;
Point point2;
Point point3;
end mypoints;

class main
  mypoints pts;
  fool f1;
end main;
```

The fields of Modelica classes are instantiated per object, i.e. a field in one object is distinct from the field with the same name in every other object instantiated from that class. The notion of class variables, i.e. variables specific to the class as opposed to instances of the class, and shared among all objects of that class, is not yet available in Modelica.

1.3.2 Initialization

Another problem is initialization of the fields. If nothing else is specified, the default start value of all numerical variables is zero, apart from function results and local variables where it is unspecified. Other start values can be specified by setting the start attribute of instance variables. For example, a start value can be specified in the class `mypoints`:

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

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

```
class main
  mypoints pts(point1.start={1,2,3});
  fool f1;
end main;
```

1.3.3 Restricted Classes

The class concept is fundamental to Modelica, and is used for a number of different purposes. Almost anything in Modelica is a class. However, in order to make Modelica code easier to read and maintain, special keywords have been introduced for specific uses of the class concept. The keywords `model`, `connector`, `record`, `block`, and `type` can be used instead of `class` under appropriate conditions. For example, a `record` is a class used to declare a record data structure and may not contain equations. A `block` is a class with fixed causality, i.e. each variable in its interface must have been declared with causality equal to `input` or `output`. A `connector` class is used to declare the structure of “ports” or interfacing points of a component and may not contain equations. A `type` is a class that can be an alias or an extension to a predefined type, record or array. For example:

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

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

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

The idea of restricted classes is beneficial since the modeler does not have to learn several different concepts, except for one: the class concept. All properties of a class, such as the syntax and semantics of definition, instantiation, inheritance, and generic properties are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified because only the syntax and semantics of the class concept has to be implemented along with some additional checks on restricted classes.

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

1.3.4 Reuse of Modified Classes

The class concept is the key to reuse of modeling knowledge in Modelica. Provisions for expressing adaptations or modifications of classes through so-called modifiers in Modelica make reuse easier. For example, assume that we would like to connect two filter models 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;
  Real u, y(start=1);
equation
  T*der(y) + y = u;
end LowPassFilter;
```

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

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

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

```
model ModifiedFiltersInSeries
```

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

1.3.5 Built-in Classes

The built-in type classes of Modelica correspond to the primitive types `Real`, `Integer`, `Boolean`, `String`, 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, e.g. `x.unit`.

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);
```

1.4 Inheritance

One of the major benefits of object-orientation is the ability to extend the behavior and properties of an existing class. The original class, known as the *superclass* or *base class*, is extended to create a more specialized version of that class, known as the *subclass* or *derived class*. In this process, the behavior and properties of the original class in the form of field declarations, equations, and other contents is 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 fields 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 fields from class `Point`, and the color fields 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;

```

1.5 Generic Classes

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

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

There are essentially two cases of generic class parameterization in Modelica: instance parameters and type parameters. Note that by class parameters in this context we do not usually mean simulation parameter variables prefixed by the keyword `parameter`, even though such variables can also be class parameters. 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.

1.5.1 Class Parameters being Components

First we present the case when class parameters are components being instances or objects. The class `C` in the example below has three class parameters marked by the keyword `replaceable`. These class parameters which are components of class `C` are declared as having the default types `GreenClass`, `YellowClass`, and `GreenClass` respectively.

```

class C
  replaceable GreenClass obj1(p1=5);
  replaceable YellowClass obj2;
  replaceable GreenClass obj3;
equation
  ...
end C;

```

Now a class `C2` is defined by providing two actual arguments `obj1` and `obj2` 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 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.

```

class C2 =
  C(redeclare RedClass obj1,
    redeclare GreenClass obj2);

```

Such a class `C2` obtained through redeclaration of `obj1` and `obj2` is of course equivalent to directly defining `C2` without reusing class `C`, as below.

```

class C2
  RedClass   obj1(p1=5);
  GreenClass obj2;
  GreenClass obj3;
equation
  ...
end C2;

```

1.5.2 Class Parameters being Types

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

```

class C
  replaceable class ColouredClass = GreenClass;
  ColouredClass      obj1(p1=5);
  replaceable YellowClass obj2;
  ColouredClass      obj3;
equation
  ...
end C;

```

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

```

class C2 =
  C(redeclare class ColouredClass = BlueClass);

```

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

```

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

```

1.6 Equations

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

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

? Assignment statements in conventional languages are usually represented as equations in Modelica.

- ? Attribute assignments are represented as equations.
- ? Connections between objects generate equations.

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

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

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

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

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

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

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

- ? Normal equations occurring in equation sections, including connect statements which is a special form of equation.
- ? Declaration equations, which are part of variable or constant declarations.
- ? Modification equations, which are commonly used to modify attributes.

Concerning attribute assignments, we have already given several examples on how to modify the start attribute of variables using equations, for example:

```
Real y(start = 1);
```

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 state variable, e.g.:

```
Real speed = 72.4;
```

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

If you need to specify an initial value for a variable, meaning its value at the start of the computation, then give an attribute equation for the start attribute of the variable, e.g.:

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

1.6.1 Repetitive Equation Structures

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, often including references to array elements denoted using square bracket notation – for more information regarding arrays see the section on arrays on page 31. 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 short hand notation for expressing a set of equations.

For example, consider an equation for a polynomial expression:

$$y = a[1] + a[2]*x + a[2]*x^1 + \dots + a[n]*x^n$$

The polynomial equation can be expressed as a set of equations with regular structure in Modelica, with y equal to the scalar product of the vectors a and $xpowers$:

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

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

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

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

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

1.7 Acausal Physical Modeling

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

for representing the physical structure of modeled systems.

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

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

1.7.1 Physical Modeling vs. Block Oriented Modeling

To illustrate the idea of acausal physical modeling we give an example of a simple electrical circuit, see Figure 1.3. The connection diagram³ of the electrical circuit shows how the components are connected and roughly corresponds to the physical layout of the electrical circuit on a printed circuit board. The physical connections in the real circuit correspond to the logical connections in the diagram. Therefore the term physical modeling is quite appropriate.

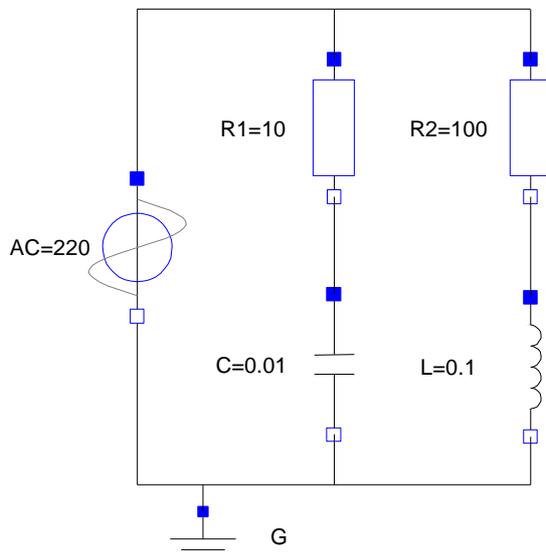


Figure 1.3 - Connection diagram of the acausal simple circuit model.

The Modelica SimpleCircuit model below directly corresponds to the circuit depicted in the connection diagram of Figure 1.3. 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 statement, which is a special syntactic form of equation that we will tell more about later.

```
model SimpleCircuit
  Resistor R1(R=10);
```

³ A connection diagram emphasizes the connections between components of a model, whereas a composition diagram specifies which components a model is composed of, their subcomponents, etc. A class diagram usually depicts inheritance and composition relations.

```

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

```

As a comparison we show the same circuit modeled using causal block oriented modeling depicted as a diagram in Figure 1.4. Here the physical topology is lost – the structure of the diagram has no simple correspondence to the structure of the physical circuit board. This model is causal since the signal flow has been deduced and is clearly shown in the diagram. Even for this simple example the analysis to convert the intuitive physical model to a causal block oriented model is non-trivial for the nonexpert. Another disadvantage is that the resistor representations are context dependent. For example the resistors R1 and R2 have different definitions, which makes reuse of model library components hard. Furthermore, such system models are usually hard to maintain since even small changes in the physical structure may result in large changes to the corresponding block oriented system model.

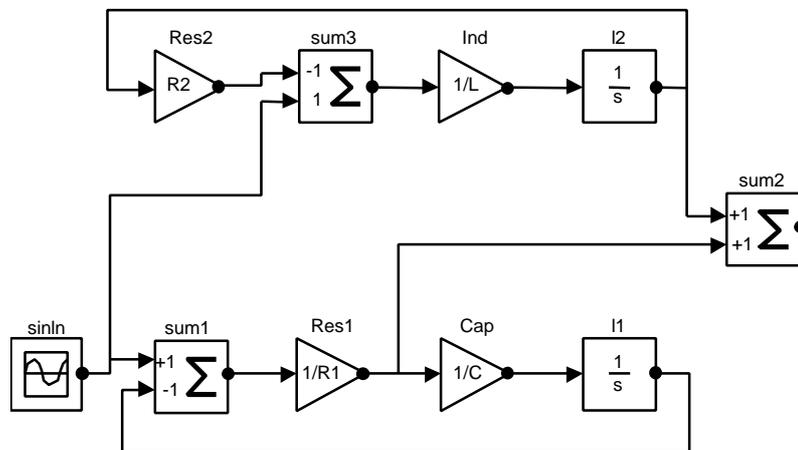


Figure 1.4 - The simple circuit model using causal block oriented modeling with explicit signal flow.

1.8 The Modelica Software Component Model

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

success in certain application areas, but there is still a long way to go to reach the level of reuse and component standardization found in hardware industry.

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

- ? Components
- ? A connection mechanism
- ? A component framework

Components are connected via the connection mechanism, which can be visualized in connection diagrams. The component framework realizes components and connections, and insures that communication works 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.

1.8.1 Components

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

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

1.8.2 Connection Diagrams

Complex systems usually consist of large numbers of connected components, of which many components can be hierarchically decomposed into other components through several levels. To grasp this complexity, a pictorial representation of components and connections is quite important. Such graphic representation is available as connection diagrams, of which a schematic example is shown in Figure 1.5. We have earlier presented a connection diagram of a simple circuit in Figure 1.3.

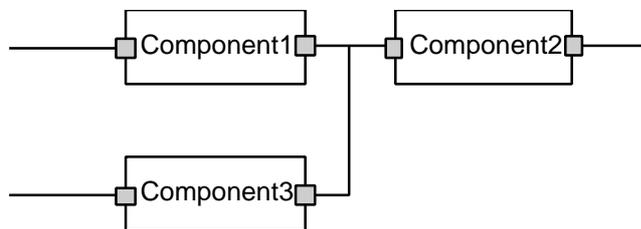


Figure 1.5 - Schematic picture of a connection diagram for components.

Each rectangle in the diagram example represents a physical component, e.g. a resistor, a capacitor, a transistor, a mechanical gear, a valve, etc. The connections represented by lines in the diagram correspond to real, physical connections. For example, connections can be realized by electrical wires, by

the mechanical connections, by pipes for fluids, by heat exchange between components, etc. The connectors, i.e. interface points, are shown as small square dots on the rectangle in the diagram. Variables at such interface points define the interaction between the component represented by the rectangle and other components.

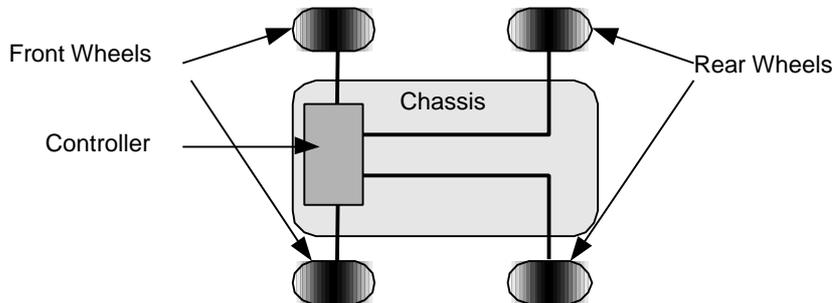


Figure 1.6 - A connection diagram for simple a car model.

A simple car example of a connection diagram for an application in the mechanical domain is shown as the car example in Figure 1.6.

The simple car model below includes variables for subcomponents such as wheels, chassis, and control unit. The wheels are connected to both the chassis and the controller. Connect statements are present, but are not shown in the partial example below.

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

1.8.3 Connectors and Connector Classes

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

For example, `Pin` is a connector class that can be used to specify the external interfaces for electrical components 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 identical to `Real`. Checking unit compatibility within equations is optional and will usually be done by special tools.

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

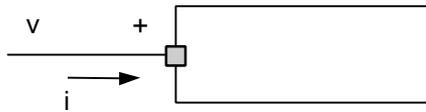


Figure 1.7 - A component with one electrical Pin connector.

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

1.8.4 Connections

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

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

- ? Equality coupling, for non-flow variables, according to Kirchhoff's first law.
- ? Sum-to-zero coupling, for flow variables, according to Kirchhoff's second 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 2:nd law.

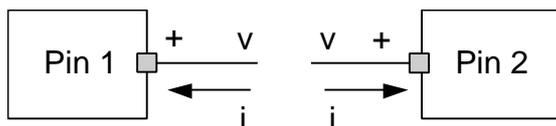


Figure 1.8 - Connecting two components that have electrical pins.

Connection statements are used to connect instances of connection classes. A connection statement `connect(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins so that they form one node. This produces two equations, namely:

```
Pin1.v = Pin2.v
Pin1.i + Pin2.i = 0
```

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

1.9 Partial Classes Describe Common Properties

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

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

The `TwoPin` class has two pins, `p` and `n`, a quantity, `v`, that defines the voltage drop across the component and a quantity, `i`, that defines the current into pin `p`, through the component, and out from pin `n`, see Figure 1.9.

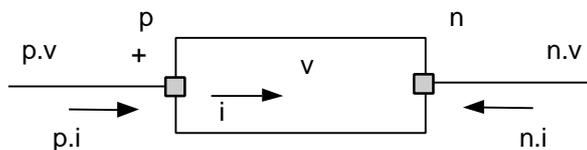


Figure 1.9 - Generic TwoPin class that describes the general structure of simple electrical components with two pins.

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.

1.9.1 Reuse of Partial Classes

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

$$R \cdot i = v;$$

This equation describes the specific physical characteristics of the relation between voltage and current for resistors.



Figure 1.10 - A resistor component.

```

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

```

A class for electrical capacitors can also reuse `TwoPin` in a similar way, adding the constitutive equation for capacitors.

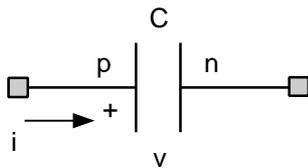


Figure 1.11 - A capacitor component.

```

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 compute the values of $i(t)$ and $v(t)$ (where t is time) so that $C v'(t)=i(t)$ for all values of t , fulfilling the constitutive equation for the capacitor.

1.10 Electrical Component Library

In a similar way as we created the resistor and capacitor components, additional electrical component classes can be created, forming a simple electrical component library. Such component libraries of reusable components are the key to effective modeling of complex systems. Below we show a small library of electrical components needed for the simple circuit example, as well as the equations that can be extracted from these components.

1.10.1 Resistor

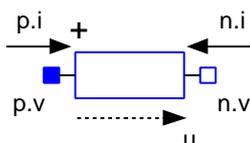


Figure 1.12 - Resistor component

Four equations can be extracted from the resistor model. The first three originate from the inherited `TwoPin` class whereas the last is the constitutive equation of the resistor.

$$\begin{aligned}
 0 &= p.i + n.i \\
 v &= p.v - n.v \\
 i &= p.i \\
 v &= R*i
 \end{aligned}$$

1.10.2 Capacitor

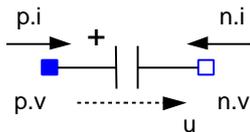


Figure 1.13 - Capacitor component

The following four equations originate from the capacitor model, where the last is the constitutive equation for the capacitor.

$$\begin{aligned}
 0 &= p.i + n.i \\
 v &= p.v - n.v \\
 i &= p.i \\
 i &= C*\text{der}(v)
 \end{aligned}$$

1.10.3 Inductor

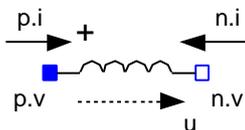


Figure 1.14 - Inductor component

The inductor class shown below gives a model for ideal electrical inductors.

```

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

```

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

$$\begin{aligned}
 0 &= p.i + n.i \\
 v &= p.v - n.v \\
 i &= p.i
 \end{aligned}$$

$$v = L \cdot \text{der}(i)$$

1.10.4 Voltage Source

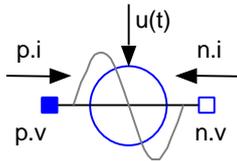


Figure 1.15 - Voltage source component `VsourceAC`, where $u(t) = VA \cdot \sin(2 \cdot \text{PI} \cdot f \cdot \text{time})$.

A class `VsourceAC` for the sin-wave voltage source to be used in our circuit example can be defined as below. This model as well as other Modelica models specify behavior that evolves as a function of time. Note that a predefined variable `time` is used which steps forward during system simulation. 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;
  input Voltage u;
equation
  v = u;
  u = VA*sin(2*PI*f*time);
end VsourceAC;
```

In this `TwoPin` based model, five equations can be extracted from the model since we introduce a special variable `u` for the voltage produced by the voltage source.

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

1.10.5 Ground

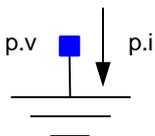


Figure 1.16 - Ground component.

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.

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

A single equation can be extracted from the Ground class.

```
p.v = 0
```

1.11 The Simple Circuit Model

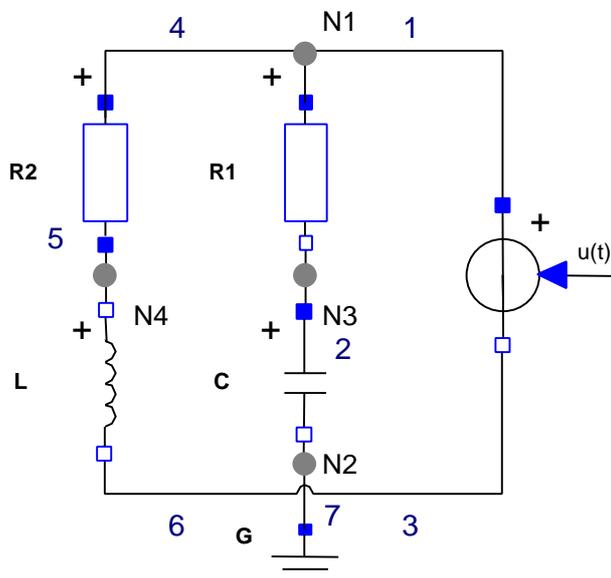


Figure 1.17 - The simple circuit model.

Having collected a small library of simple electrical components we can now put together the simple electrical circuit shown previously. The two resistor instances R1 and R2 are declared with modification equations for their respective resistance parameter value. 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. Connection statements are provided to connect the components in the circuit.

```
class SimpleCircuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect (AC.p, R1.p); // 1, Capacitor circuit
```

```

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

```

1.12 Arrays

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

```

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

```

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

```

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

```

In the first two array declarations, declaration equations have been given, where the array constructor `{}` is used to construct array values for defining `positionvector` and `identitymatrix`. Indexing of an array `A` is written `A[i,j,...]`, where 1 is the lower bound and `size(A,k)` is the upper bound of the index for the `k`:th dimension. Submatrices can be formed by utilizing the `:` notation for index ranges, for example `A[i1:i2, j1:j2]`. Array expressions can be formed using the arithmetic operators `+`, `-`, `*`, and `/`, since these can operate on either scalars, vectors, matrices, or (when applicable) multi-dimensional arrays with elements of type `Real` or `Integer`. The multiplication operator `*` is scalar product when used between vectors, matrix multiplication when used between matrices, and element-wise multiplication when used between an array and a scalar. As an example, multiplying `positionvector` by the scalar 2:

```
positionvector*2
```

which gives the result:

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

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

<code>transpose(A)</code>	Permutes the first two dimensions of array <code>A</code>
---------------------------	---

<code>zeros(n₁,n₂,n₃,...)</code>	Returns an $n_1 \times n_2 \times n_3 \times \dots$ zero-filled integer array.
<code>ones(n₁,n₂,n₃,...)</code>	Returns an $n_1 \times n_2 \times n_3 \times \dots$ one-filled integer array.
<code>fill(s,n₁,n₂,n₃, ...)</code>	Returns the $n_1 \times n_2 \times n_3 \times \dots$ array with all elements filled with the value of the scalar expression s .
<code>min(A)</code>	Returns the smallest element of array expression A .
<code>max(A)</code>	Returns the largest element of array expression A .
<code>sum(A)</code>	Returns the sum of all the elements of array expression A .

A scalar function can be applied to arrays element-wise, e.g. 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,\dots along the k :th dimension.

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 semicolon and columns by comma. The example below creates an $m \times n$ matrix:

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

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

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

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

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

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

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

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

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

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

```
[1]
```

gives the matrix:

```
{{1}}
```

1.13 Algorithmic Constructs

Even though equations are eminently suitable for modeling physical systems, and for a number of other tasks, there are situations where non-declarative algorithmic constructs are needed. This is typically case for algorithms, i.e. stepwise procedural descriptions on how to carry out specific computations.

1.13.1 Algorithms

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

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

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

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

```

u = x1+x2;
...

```

Note that the code in the algorithm section, sometimes denoted algorithm equation, uses the values of certain variables from outside the algorithm. These variables are so called input variables to the algorithm – in this example x, y and z . Analogously, variables assigned values by the algorithm define the outputs of the algorithm – in this example x_1 and x_2 . 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.

1.13.2 Statements

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

```

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

```

Consider once more the computation of the polynomial presented in the section on repetitive equation structures on page 20.

```

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

```

1.13.3 Functions

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

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

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

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

```
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 multi-result functions may only be called 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);
```

1.13.4 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;

```

1.13.5 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;

```

1.14 Hybrid Modeling

Macroscopic physical systems in general evolve continuously as a function of time, obeying the laws of physics. This includes the movements of parts in mechanical systems, current and voltage levels in

electrical systems, chemical reactions, etc. Such systems are said to have continuous dynamics.

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

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

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

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

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 1.18, and the ideal diode characteristic in parameterized form is shown in Figure 1.19.

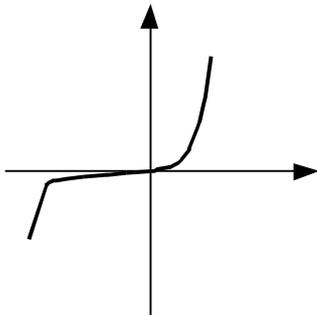


Figure 1.18 - Real diode characteristic.

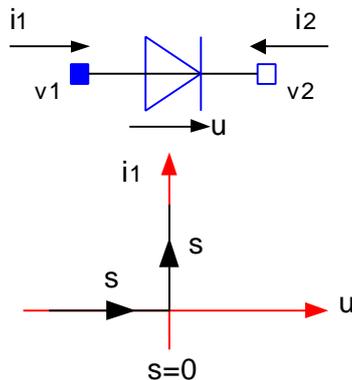


Figure 1.19 - Ideal diode characteristic.

Since the voltage level of the ideal diode would go to infinity in an ordinary voltage-current diagram, a parameterized description is more appropriate, where both the voltage v and the current i 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
  i = if off then 0 else s; // conditional expression
end Diode;

```

When-clauses have been introduced in Modelica to express instantaneous equations, i.e. equations that are valid only at certain points, e.g. at discontinuities, when specific conditions become true. The syntax of when-clauses for the case of a vector of conditions is shown below. The equations in the when-clause are activated when at least one of the conditions become true. 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-clause is appropriate when modeled. The motion of the ball is characterized by the variable `height` above the ground and the vertical velocity `v`. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 1.20. When the ball bounces against the ground its velocity is reversed. An ideal ball would have an elasticity coefficient of 1 and would not lose any energy at a bounce. A more realistic ball, as the one modeled below, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

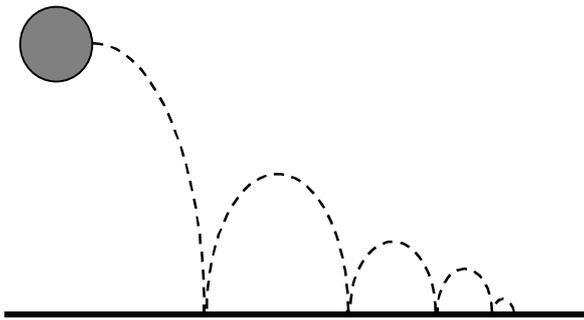


Figure 1.20 - A bouncing ball.

The bouncing ball model contains the two basic equations of motion relating height and velocity as well as the acceleration caused by the gravitational force. At the bounce instant the velocity is suddenly reversed and slightly decreased, i.e. $v(\text{after bounce}) = -c \cdot v(\text{before bounce})$, which is accomplished by the special syntactic form of instantaneous equation: `reinit(v, -c*v)`.

```

model BouncingBall "The bouncing ball model"
  parameter Real g=9.81; // gravitational acceleration
  parameter Real c=0.90; // elasticity constant of ball
  Real height(start=0); // height above ground
  Real v(start=10); // velocity
equation
  der(height) = v;
  der(v) = -g;
  when height<0 then
    reinit(v, -c*v);
  end when;
end BouncingBall;

```

1.15 Packages

Name conflicts is 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 you are using short descriptive names since such names are easy to use and therefore quite popular, making them quite likely to be used in another person's code.

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

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

Modelica has defined the package concept as a restriction and enhancement of the class concept. Thus, inheritance could be used for importing definitions into the name space of another package. However, this

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

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

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

end mypack;
```

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

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

Another disadvantage with the above package is that the `Ampere` type is referred to using standard nested lookup and not via an explicit `import` statement. Thus, in the worst case you may have to visually scan the whole source code of a large package in order to find all such dependences. Instead, a well-designed package should state all its dependences explicitly through `import` statements which are easy to find. You 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, insuring that all dependences 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.

See Chapter **Error! Reference source not found.** for additional details concerning packages and import.

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

class foo;
  SI.Voltage v;
  Modelica.SIunits.Ampere i;
end foo;
```

```
end mypack;
```

1.16 Implementation and Execution of Modelica

In order to gain a better understanding of how Modelica works it is useful to take a look at the process of translation and execution of a Modelica model, which is sketched in Figure 1.21. First the Modelica source code is parsed and converted into an internal representation, usually an abstract syntax tree. This representation is analyzed, type checking is done, classes are inherited and expanded, modifications and instantiations are performed, connect statements are converted to 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 dependences between the equations. In the case of general differential algebraic equations (DAEs), this is not just sorting, but also manipulation of the equations to convert the coefficient matrix into block lower triangular form, a so-called BLT transformation. Then an optimizer module containing algebraic simplification algorithms, symbolic index reduction methods, etc., eliminates most equations, only keeping a minimal set that eventually will be solved numerically. As a trivial example, if two syntactically equivalent equations appear, only one copy of the equations is kept. Then independent equations in explicit form are converted to assignment statements. This is possible since the equations have been sorted and an execution order has been established for evaluation of the equations in conjunction with the iteration steps of the numeric solver. If a strongly connected set of equations appears, this set is transformed by a symbolic solver which performs a number of algebraic transformations to simplify the dependencies between the variables. It can sometimes solve a system of differential equations if it has a symbolic solution. Finally, C code is generated, and linked with a numeric equation solver that solves the remaining, drastically reduced, equation system.

The approximations to initial values are taken from the model definition or are interactively specified by the user. If necessary, the user also specifies the parameter values. A numeric solver for differential-algebraic equations 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 $\mathbb{R}^2.v(\tau)$ in the simple circuit model. Those functions can be displayed as a graphs and/or saved in a file.

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

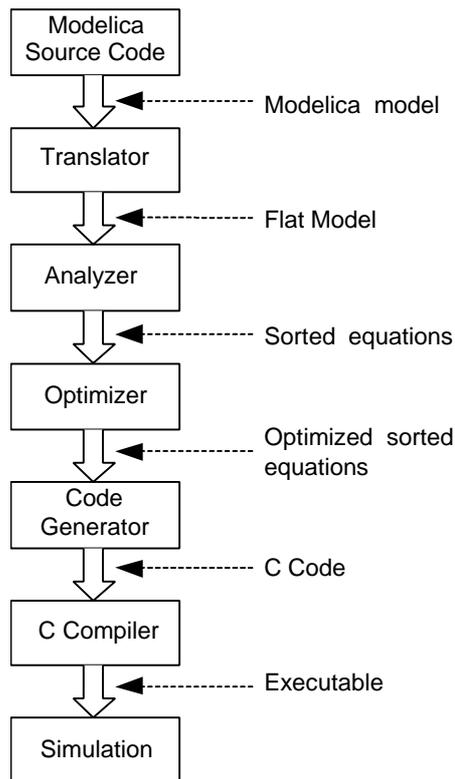


Figure 1.21 - The stages of translating and executing a Modelica model.

1.16.1 Hand Translation of the Simple Circuit Model

Let us return once more to the simple circuit model, previously depicted in Figure 1.3, but for the reader's convenience also shown below in Figure 1.22. It is instructive to translate this model by hand, in order to understand the process.

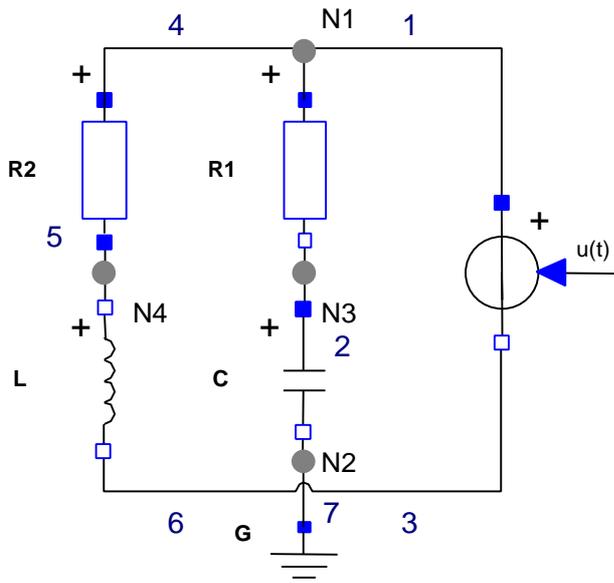


Figure 1.22 - The simpleCircuit model once more.

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

- ? 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 (ODE) -- both alternatives are possible, since a DAE normally can be transformed into an ODE.
- ? For each connection between instances within the model, add connection equations to the DAE system so that non-flow 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 labelled 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 non-flow variable representing voltage potential. Therefore, the equality equation $R1.p.v=AC.p.v$ is generated. Equality equations are always generated when non-flow variables are connected.

Notice that another wire (labelled 4) is attached to the same pin, R1.p. This is represented by an additional connect statement: `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 1) generated from the SimpleCircuit class consists of 33 differential-algebraic equations. These include 33 variables, as well as time and several parameters and constants.

AC	0 = AC.p.i+Ac.n.i	L	0 = L.p.i+L.n.i
	AC.v = Ac.p.v-AC.n.v		L.v = L.p.v-L.n.v

	$AC.i = AC.p.i$ $AC.v = AC.u$ $AC.u = AC.VA * \sin(2 * AC.PI * AC.f * time);$		$L.i = L.p.i$ $L.v = L.L * \text{der}(L.i)$
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	$0 = C.p.i + C.n.i$ $C.v = C.p.v - C.n.v$ $C.i = C.p.i$ $C.i = C.C * \text{der}(C.v)$	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 1 - The equations extracted from the simple circuit model – an implicit DAE system.

Below are the 33 variables in the system of equations, of which 31 are algebraic variables. Two variables, C.v and L.i, are state variables since their derivatives occur in the equations.

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
AC.u	G.p.i	G.p.v		

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

1.16.2 Transformation to State Space Form

The implicit differential algebraic system of equations (DAE system) in Table 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:

- ? All constant variables which are simulation 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.
- ? Variables declared with the input attribute, i.e. prefixed by the `input` keyword, that appear in

instances at the highest hierarchical level, are collected into an input vector u .

- ? Variables whose derivatives appear in the model, i.e. the `der()` operator is applied to those variables, are collected into a state vector x .
- ? 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:

$$p = \{R1.R, R2.R, C.C, L.L, AC.VA, AC.f\}$$

$$u = \{AC.u\}$$

$$x = \{C.v, L.i\}$$

$$y = \{R1.p.i, R1.n.i, R1.p.v, R1.n.v, R1.v, R1.i, R2.p.i, R2.n.i, R2.p.v, R2.n.v, R2.v, R2.i, C.p.i, C.n.i, C.p.v, C.n.v, C.i, L.n.i, L.p.v, L.n.v, L.v, AC.p.i, AC.n.i, AC.p.v, AC.n.v, AC.i, AC.u, AC.v, G.p.i, G.p.v\}$$

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

$$x' = f(x, t)$$

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

For the simple circuit model we have the following:

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

$$x' = \{\text{der}(C.v), \text{der}(L.i)\}$$

1.16.3 Solution Method

We will use an iterative numerical solution method. First, assume that estimated values 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 x' (i.e. $\text{der}(x)$) at time t , e.g.:

$$\text{der}(x) = (x(t+h) - x(t))/h$$

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

$$x(t+h) = x(t) + \text{der}(x) * h$$

In this way the value of the state vector x is computed one step ahead in time for each iteration, provided $\text{der}(x)$ can be computed at the current point in simulated time. However, the derivative $\text{der}(x)$ of the

state vector can be computed from $x'=f(x,t)$, i.e. by selecting the equations involving $\text{der}(x)$, and algebraically extracting the variables in the vector x in terms of other variables, as below:

$$\begin{aligned}\text{der}(C.v) &= C.i/C.C \\ \text{der}(L.i) &= L.v/L.L\end{aligned}$$

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 (1) through (3) below.

$$\begin{aligned}C.i &= R1.v/R1.R \quad (1) \\ \text{using: } C.i &= C.p.i = -R1.n.i = R1.p.i = R1.i = R1.v/R1.R\end{aligned}$$

$$\begin{aligned}R1.v &= R1.p.v - R1.n.v = R1.p.v - C.v \quad (2) \\ \text{using: } R1.n.v &= C.p.v = C.v+C.n.v = C.v+AC.n.v = C.v+G.p.v = C.v+0 = C.v\end{aligned}$$

$$\begin{aligned}R1.p.v &= AC.p.v = AC.VA*\sin(2*AC.f*AC.PI*t) \quad (3) \\ \text{using: } AC.p.v &= AC.v+AC.n.v = AC.u+G.p.v = AC.VA*\sin(2*AC.f*AC.PI*t)+0\end{aligned}$$

In a similar fashion we derive equations (4) and (5) below:

$$\begin{aligned}L.v &= L.p.v-L.n.v = R1.p.v - R2.v \quad (4) \\ \text{using: } L.p.v &= R2.n.v = R1.p.v-R2.v \\ \text{and: } L.n.v &= C.n.v = AC.n.v = G.p.v = 0\end{aligned}$$

$$\begin{aligned}R2.v &= R2.R*L.p.i \quad (5) \\ \text{using: } R2.v &= R2.R*R2.i = R2.R*R2.p.i = R2.R*(-R2.n.i) = R2.R*L.p.i \\ &= R2.R*L.i\end{aligned}$$

Collecting the five equations together:

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

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

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

```
der(C.u) := C.i/C.C
```

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

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

It should be emphasized that the simple circuit example is trivial. Realistic simulation models often contain tens of thousands of equations, non-linear 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.

<pre>delay(expr, delayTime, delayMax) delay(expr, delayTime)</pre>	<p>Returns the value of <i>expr</i> at the time (<i>time-delayTime</i>) for <i>time</i> > <i>time.start</i> + <i>delayTime</i> and the value of <i>expr</i> at the time <i>time.start</i> for <i>time</i> <= <i>time.start</i>+<i>delayTime</i>. The arguments, i.e., <i>expr</i>, <i>delayTime</i> and <i>delayMax</i>, need to be subtypes of Real. The expression <i>delayMax</i> must be a parameter expression, and is used to avoid excessive storage requirements and enhance efficiency by storing the delayed expression for a maximum delay time given by <i>DelayMax</i>.</p> <p>The following relation must hold: $0 \leq \textit{delayTime} \leq \textit{delayMax}$, otherwise an error occurs. If <i>delayMax</i> is not supplied in the argument list, <i>delayTime</i> needs to be a parameter expression.</p>
<pre>cardinality(c)</pre>	<p>Returns the number of (internal and external) occurrences of connector instance <i>c</i> in connect statements of the current model as an Integer number.</p> <p>For a detailed example on the use of <i>cardinality</i>, see the section on connection dependent connection equations on Error! Bookmark not defined. in Chapter 4.</p>
<pre>direction(c)</pre>	<p>The function <i>direction(c)</i> gives information about the direction of the connection, provided that <i>cardinality(c)</i> == 1, i.e. the connector <i>c</i> is involved in exactly one connection.</p> <p>For a connection clause <i>connect(c1, c2)</i> fulfilling that constraint, <i>direction(c1)</i> returns -1 and <i>direction(c2)</i> returns 1.</p> <p>See also Section Error! Reference source not found. on page Error! Bookmark not defined.</p>

1.16.4 Event Iteration and the Pre Function

Regarding *event iteration*, a new event is triggered automatically if at the time instant t of the current event it is the case that for at least one variable v , “ $\text{pre}(v) \neq v$ ” after the active model equations have been evaluated. Typically a discontinuity causes the value of a discontinuous variable v to be different from its predecessor value $\text{pre}(v)$. In this case the model equations are reevaluated, until for all variables v used in pre -operators it is the case that “ $\text{pre}(v) = v$ ”. This evaluation sequence is called *event iteration*. After such an evaluation sequence the solution process for the continuous part of the equation system is resumed, also known as “continuous integration”. More detailed information regarding this subject is available in Chapter ??.
