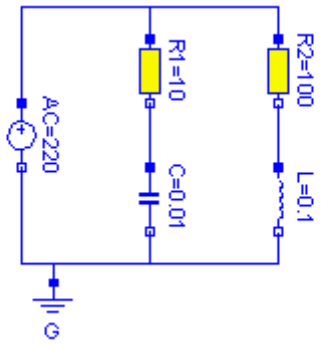
1. Modelica at a Glance

To give an introduction to Modelica we will consider modeling of a simple electrical circuit as shown below.



The system can be broken up into a set of connected electrical standard components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of these components are typically available in model libraries and by using a graphical model editor we can define a model by drawing an object diagram very similar to the circuit diagram shown above by positioning icons that represent the models of the components and drawing connections.

A Modelica description of the complete circuit looks like

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

equation
  connect (AC.p, R1.p);     // Capacitor circuit
  connect (R1.n, C.p);
  connect (C.n, AC.n);
  connect (R1.p, R2.p);     // Inductor circuit
  connect (R2.n, L.p);
  connect (L.n,  C.n);
  connect (AC.n, G.p);      // Ground
end circuit;
```

For clarity, the definition of the graphical layout of the composition diagram (here: electric circuit diagram) is not shown, although it is usually contained in a Modelica model as annotations (which are not processed by a Modelica translator and only used by tools). A composite model of this type specifies the topology of the system to be modeled. It specifies the components and the connections between the components. The statement

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

declares a component `R1` to be of class `Resistor` and sets the default value of the resistance, `R`, to 10. The connections specify the interactions between the components. In other modeling languages connectors are referred as cuts, ports or terminals. The language element **connect** is a special operator that generates equations taking into account what kind of quantities that are involved as explained below.

The next step in introducing Modelica is to explain how library model classes are defined.

A connector must contain all quantities needed to describe the interaction. For electrical components we need the quantities voltage and current to define interaction via a wire. The types to represent them are declared as

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

where `Real` is the name of a predefined variable type. A real variable has a set of attributes such as unit of measure, initial value, minimum and maximum value. Here, the units of measure are set to be the SI units.

In Modelica, the basic structuring element is a **class**. There are seven *restricted* classes with specific names, such as **model**, **type** (a class which is an extension of built-in classes, such as **Real**, or of other defined types), **connector** (a class which does not have equations and can be used in connections). For a valid model it is fully equivalent to, e.g., replace the **model**, and **type** keywords by the keyword **class**, because the restrictions imposed by such a specialized class are fulfilled by a valid model.

The concept of restricted classes is advantageous because the modeler does not have to learn several different concepts, but just one: the class concept. All properties of a class, such as syntax and semantic of definition, instantiation, inheritance, genericity are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified considerably because only the syntax and semantic of a **class** has to be implemented along with some additional checks on restricted classes. The basic types, such as `Real` or `Integer` are built-in **type** classes, i.e., they have all the properties of a class and the attributes of these basic types are just parameters of the class.

There are two possibilities to define a class: The standard way is shown above for the definition of the electric circuit (**model** circuit). A short hand notation is possible, if a new class is identical to an existing one and only the default values of attributes are changed. The types above, such as *Voltage*, are declared in this way.

A connector class is defined as

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

A connection **connect** `(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, namely `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current 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 flow rates in a piping network and to forces and torques in mechanical systems.

When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes. A common set of connector classes used in all components in the library supports compatibility of the component models. In the Modelica Standard Library developed together with the Modelica Language, for many domains appropriate connector definitions are already available.

A common property of many electrical components is that they have two pins. This means that it is useful to define an "interface" model class,

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

that 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 the pin `p`, through the component and out from the pin `n`. The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added. The keyword **`partial`** indicates that this model class is incomplete. The key word is optional. It is meant as an indication to a user that it is not possible to use the class as it is to instantiate components. Between the name of a class and its body it is allowed to have a string. It is treated as a comment attribute and is meant to be a documentation that tools may display in special ways.

To define a model for a resistor we exploit OnePort and add a definition of parameter for the resistance and Ohm's law to define the behavior:

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

The keyword **parameter** specifies that the quantity is constant during a simulation run, but can change values between runs. A parameter is a quantity which makes it simple for a user to modify the behavior of a model.

A model for an electrical capacitor can also reuse the TwoPin as follows:

```
model Capacitor "Ideal electrical capacitor"
  extends OnePort;
  parameter Real C(unit="F") "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```

where **der**(v) means the time derivative of v. A model for the voltage source can be defined as

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

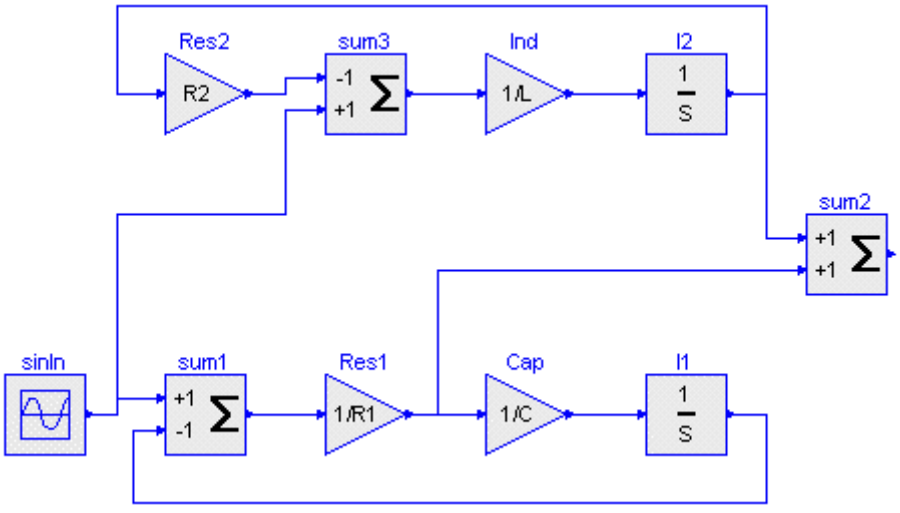
In order to provide not too much information at this stage, the constant PI is explicitly declared, although it is usually imported from the Modelica standard library (see appendix of the Language Specification). Finally, we must not forget the ground point.

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

The purpose of the ground model is twofold. First, it defines a reference value for the voltage levels. Secondly, the connections will generate one Kirchhoff's current law too many. The ground model handles this by introducing an extra current quantity p.i, which implicitly by the equations will be calculated to zero.

Comparison with block oriented modeling

If the above model would be represented as a block diagram, the physical structure will not be retained as shown below. The block diagram is equivalent to a set of assignment statements calculating the state derivatives. In fact, Ohm's law is used in two different ways in this circuit, once solving for i and once solving for u.



This example clearly shows the benefits of physically oriented, *non-causal modeling* compared to block oriented, causal modeling.