Object Oriented Simulation using Modelica

Stefan Andersson¹, Johan Malmström², Mattias Tidlund³

Course: CD5130 Object Oriented Development, Advanced Course
Student Advisor: Mikael Sandberg
Department of Computer Engineering
Mälardalen University, P.O. Box 883, 721 23 Västerås, Sweden
¹dal98san@mds.mdh.se
²dal98jmm@mds.mdh.se
³mtd98005@student.mdh.se

Abstract:
In this report we present the new standard tool of object oriented simulation, the object oriented programming language Modelica. Modelica is a modern language built on non-casual modeling with mathematical equations and object oriented constructs to facilitate reuse of modeling knowledge. The design of Modelica started in the continuous time domain since there is a common mathematical framework in the form of differential-algebraic equations. We have also studied the modeling and simulation environment Dymola, which is the first simulation environment for the Modelica language. Dymola is a product from the Dynasim company.

The report contains an introduction to the Modelica language and its design from a computer science point of view.

1 Introduction

The purpose of this paper is to introduce the reader to object oriented simulation in general and particular to the programming language Modelica, a new standard tool in modeling. In our paper the reader gets a brief introduction to object oriented simulation using Modelica. We have studied the language from a computer science point of view in the absence of engineering skills. The reader will be guided through some simple modeling examples in Modelica.

2 Introduction to object oriented simulation

With object oriented simulation (OOS) the way of creating simulation software design changed. Using objects the modeling of real world entities became better than the traditional software modules. Having a proper OOS framework simulation can be made efficient to develop and maintain. For that reason OOS has become the state of the art in simulation design. OOS started with the development of object oriented languages and the first language that was the beginning of OOS was SIMULA. With languages like Smalltalk the growth of OOS started and modern language like C++ are widely used to design OOS software. Today object oriented simulation software is starting to dominate the simulation software market.
As in object oriented programming (OOP) the basic component in OOS is an object. These objects can be looked at as data structures with their own private variables and methods, with no knowledge about other objects. The basic concepts of OOS is similar to OOP with classes, inheritance, polymorphism, communication between objects by sending messages, dynamic binding, etc.

The programmer has to detect the basic entities for the domain to be modeled, to create corresponding classes which would represent them in a simulation system and have a plan for how an object should work with other objects. The making of an OOS design is different from the traditional procedural approach, ”separate procedures on separate data” [3].

OOS fits well into area of object oriented (OO) software design, but it requires some special simulation techniques, well-known software engineering techniques are not always applied to for designing simulation software. There are three standard software design features, reusability, adaptability and maintainability, that should be supported by any simulation environment for separation of physical, control and information elements of a given system. In an OOS-based environment these design features and other software designs should be supported.

As an example, in an implementation of an event-driven simulation approach, a few major concepts that are needed in OOS and it shows a few differences between OOP and OOS [3]:

- Entities are active objects.
- A change of an objects state is represented as an event and thus synchronizes actions between two entities or passes messages between them. A consequence of this is that a event-list management has to be considered in every kind of design that supports this kind of behavior.
- Simulation time uses a logical clock that is updated by events.
- There has to be different data structures for starting, conducting and terminating simulation.

Methods for standard OO analysis and design can also be applied to OOS design. Even some of the methods which are applied to OOP-design can also be applied to OOS. And there also exists several specific methods for OOS design.
3 Modelica

3.1 Why Modelica?

Modeling and simulation are becoming more important since engineers need to analyze increasingly complex systems often composed of subcomponents from different domains. Typical examples come from mechatronic systems within automotive, aerospace and robotics applications. These systems are composed of components from domains like electrical, mechanical, hydraulic, control, etc. The existing tools are in general weak handling multi-domain models because the general tools are block-oriented and it takes a lot of rewriting to get the equations into explicit forms.

Between the user’s problem and the model description that the simulation program understands there is a large gap, modeling should be much closer to the way an engineer builds a real system first trying to find standard components with appropriate specifications and interfaces. The main problem is the absence of a state-of-the-art, standard external representation. Modeling languages don’t support structuring of large complex models and the process of model evolution in general.

There are two concepts among the recent research results in modeling and simulation that have a strong relevance to the problem [2]:

- Object-oriented modeling languages that have already proved to be successfully used to support hierarchical structuring, reuse and evolution of large complex models independent from the application domain and specialized graphical formalisms.

- Non-causal modeling shows that the traditional simulation abstraction with input/output block can be generalized by not committing ports to an input/output role early. By doing this it makes it easier to enable more simple models and more efficient simulation and still have the capability to include sub models with fixed input/output roles.

3.2 Alternatives to Modelica

There are at least four [2] alternatives to Modelica, namely:

**General purpose simulation tools** that are already established. For example commercial tools as SCSL, EASY5 and SIMULINK. Modelica will have to offer significant practical advantages to these.

**Special purpose simulation programs.** There already exists a wide variety of simulation tools and programs with strong model libraries for specific domains, i.e. for electronics, multi-body systems and chemical processes. Modelica’s advantage to these is their lack of multi-domain simulation.

**Numerical subroutine libraries and traditional programming languages.** Many industrial simulation studies are still done without the use of any general purpose simulation tools.

**VHDL-AMS** is a IEEE supported alternative language standardization effort underway.
4 Introduction to Modelica

Modelica is a language for modeling of physical systems designed to support effective library and model exchange. It is a modern language built on non-causal modeling with mathematical equations and object oriented construct to facilitate reuse of modeling knowledge.

In September 1996 a design group started within the ESPRIT project. The group, called ”Simulation in Europe Basic Research Working Group (SiE-WG)”, was formed with people from different application domains, simulation tool builders and computer scientists. Their purpose was to create a modeling language that has the ability to reuse and exchange models on a standardized format. In an attempt to combine existing modeling languages combined with object oriented modeling- and non-causal modeling languages one year later the Modelica language for modeling was accomplished. Modelica is a language for many areas of applications such as electrical circuits, multi-body systems, drive trains, hydraulics, thermo-dynamical systems, etc. It also uses several formalisms: ordinary differential equations (ODE), differential-algebra equations (DAE), bond-graphs, finite state automata and Petri nets, etc. The group is currently working as Technical Committee 1 within Eurosim and Technical Chapter within Society Computer Simulation International.

Society Computer Simulation International is a principal society devoted to the advancement of simulation and allied computer arts in all fields. The main purpose is to create communications between professionals in the field of simulation.

The purpose of Modelica was to create a language that serves as a standard format for users and tools from different domains to exchange models with each other. In most cases the Modelica syntax is hidden from the end-user, because a graphical user interface is used to build models by selecting icons for model components using dialogue boxes for parameter entry and connecting components graphically. To create Modelica the work started in the continuous time domain because there is a common framework in the form of differential-algebra equations with several existing modeling languages based on similar ideas. These languages had been used in various applications, therefore the work started to collect all knowledge and experience about these languages in order to create the design of Modelica.

The design groups first goal was to design a modeling language for differential-algebra equations systems with discrete event features to handle discontinuities and sampled systems. This design had the feature to be extendible so the design could be expanded to handle multi-formalism, multi-domain and general-purpose modeling languages. Modelica is created to be a de facto standard for representing models and to support model exchange. The two concepts object oriented and non-causal modeling was a strong influence to the Modelica design.

4.1 Advanced Modeling Features

The modeling possibilities of Modelica are many. Here are some of the more powerful constructs Modelica supports:

Vectors, matrices and arrays. Multi-body systems and control systems is conveniently done by using matrix equations, multi-dimensional matrices and usual matrix operators and matrix functions are supported in Modelica. It is also possible to define regular connection patterns and to use arrays of components.

Class parameters. The reuse of Modelicas model library components also supports model class parameter. For example if we would like to replace a controller with some other controller, it is possible just to replace the controller in a graphical user environment, i.e. to create a new model. The problem with this solution is that two models must be maintained. Modelica has instead
the capability to substitute the model class of certain components using a language construct at
the highest hierarchical level, so only one version of the rest of the model is needed.

The new component must be a subtype of the replaced component, i.e. it must have compatible
connectors and parameters. The type system in Modelica is greatly influenced by type theory,
in particular the notion of subtyping which is different from subclassing. Flexibility is the main
benefit in the composition of types, while maintaining a rigorous type system. Inheritance is
not used for classification and type checking in Modelica.

In a real applications there could be many controllers of the same type, so the approach above
could be a bit clumsy because we need to know the names of all controllers. To avoid this
problem and prepare for replacement of a set of models, a replaceable class can be defined.

Hybrid modeling. Realistic physical modeling often contain discontinuities, discrete events or change
of structure for example relays, switches, friction, impact, sampled data systems, etc. In Modelica
there is a special language constructs that allows simulators to handle such events. Special
design emphasis was given to synchronization and propagation of events and possibility to find
consistent restarting conditions after an event. It is also possible to model automatic gear boxes
for the purpose of real-time simulation. Modelica even supports development of model libraries
for finite state machines and Petri nets.

Algorithm and functions. For modeling a part of a system in procedural programming Modelica
supports algorithm and functions. The syntax is similar to other Modelica classes and matrix
expressions can be used. Assignment-statements, if-statements and loops can be used as usual.

Standard Libraries. For Modelica to be useful for model exchange, it is important that libraries of
the most commonly used components are available, ready to be used and sharable between
applications. For that reason an extensive Modelica base library is under development which
will become a part of Modelica. The library will include mathematical functions (sin, ln, etc.),
type definitions (e.g. Angle, Voltage), interface definitions (e.g. Pin, Flange) and component
libraries for various domains [4].

Predefined quantity types and connectors are useful for standardization of the interfaces be-
tween components and achieve model compatibility without having to resort to explicit coordi-
nation of modeling activities.

Component libraries are mainly derived from already existing model libraries from various
object oriented modeling systems. They are created by specialists in respective area, taking
advantage of the new features of Modelica not available in the original modeling system.
5 Design of Modelica

5.1 Design goals

The following nine general principles and design goals have been applied during the design of the Modelica language [2].

- Engineering tool
- Reliability and correctness
- Coping with system evolution
- Generality, uniformity
- Declarativity and referential transparency
- Adherence to common de facto language standards
- High level of abstraction
- Code reuse
- Mathematical foundation

5.1.1 Engineering tool

To design Modelica as an engineering tool the Modelica designers studied different simulation domains, e.g. electrical circuits, multi-body and energy domain systems. All these domains have specific requirements that have to be fulfilled by Modelica, but they also have common requirements such as allowing efficient implementation and coping with large physical systems composed of different kind of subsystems.

Modelica is intended to be a standard tool in these different domains so models can be exchanged between tools and users [6].

5.1.2 Reliability and correctness

Modelica has very strong typing, this is to provide partial verification of internal consistency. The language uses named parameter passing to improve the reliability that is necessary in a engineering tool. One of the most fundamental goals in the design of Modelica is to support construction of reliable and correct software.

5.1.3 Coping with system evolution

For a engineering tool its important that systems can evolve. It should be easy to add new functionality, adapting to new hardware, enhancing performance, etc. Another aspect of system evolution is control over system complexity. Modelicas way of coping with system evolution is its strong type system. This strong type system leads to a partially verified system at each stage, and combined with Modelicas class and package concept system complexity is handled [1].

Modelicas type system is greatly influenced by the type theory introduced by Abadi and Cardelli [2]. Abadi and Cardelli separate the notion of sub classing from the notion of sub typing. The main benefit is added flexibility in the composition of types.
5.1.4 Generality, uniformity

Modelica uses classes for declaring objects. A class contains a list of component declarations and a list of equations. To improve readability and maintenance, specific keywords have been introduced for specific classes; model, connector, record, block, type and package. These keyword are restricted versions of the general class concept, so for a valid model, the special class keywords could be replaced by the keyword class and give exactly the same model behavior. The built-in types are Real, Boolean, Integer, String.

Object oriented features like multiple inheritance, sub typing, and parametric polymorphism is integrated through the general static and strong type system of Modelica.

5.1.5 Declarativity and referential transparency

Modelica functions are declarative and encourage a functional programming style. They are essentially side effect free mathematical functions. The use of equations and not assignment statements gives the flexibility needed e.g. that a component description can be used with different causalities depending on how the component is connected. The body of a function is called an algorithm section, which can be regarded as a strongly connected set of equations.

5.1.6 Adherence to common de facto language standards

There are many programming language standards used today in the area of simulation. In order to be easier to learn and use for engineers Modelica tries to be partly compatible to some of these programming languages.

Some other standards adopted is the UniCode character standard1, some Java syntax and Modelica uses the Matlab notation for matrix operations.

5.1.7 High level of abstraction

Modelica is a specification language, e.g. a language with modularized and parameterized mechanisms. Modelica is designed to allow abstraction from unnecessary details.

This abstraction is obtained by equations integrated with object oriented structure concepts and object connection mechanisms.

5.1.8 Code reuse

Code reuse is a very important goal in software development today. Modelica reaches this goal by:

Non-casual equations permits model components to be reused in different contexts.

Inheritance of types by extending predefined models.

Polymorphism.

Code reuse improves and optimizes the development of software.

1http://www.unicode.org/
5.1.9 Mathematical foundation

The design of Modelica started in the continuous time domain since there is a common mathematical framework in the form of DAEs [5]. The result of this is a language with a strong mathematical foundation in the sense that a Modelica model is expanded into a set of DAE. Equations can be conditional, to represent discrete-event features and enable hybrid modeling.
6  A overview of the Modelica language

The basic structure element of Modelica is a class. There are seven restricted classes with specific names, model, type and connector are examples of restricted classes. The purpose with restricted classes is that the modeler doesn’t have to learn only one concept, the class concept. All the properties of a class are identical to all kinds of restricted classes.

6.1  Basic Language elements

Basic types are such as Real, Integer, Boolean and String and they are built in type classes, with all the properties of a class and the attributes are just parameters of the class. With a specifier like parameter or constant, we give a component a constant value and the value will keep being constant during simulation run-time. It can be changed when a component is reused or between simulation runs.

\[
\text{Real } v, y(\text{start}=1);
\text{parameter Real } U=1;
\]

The Real variable has an attribute called start to give it’s initial value.

6.2  Classes for reuse

A class declaration contains a list of components and a list of equations. The keyword equation is used to declare the list of equations. Equations can have expressions on both on the right and on the left hand side. The class concept is used for many purposes and are similar to the class concept in programming languages. Modelica has special keywords for special uses, model, connector, record, block, type, package, to make it easier to read and maintain. It is possible to replace these keywords in a valid model with class and it would give exactly the same model behavior.

\[
\text{class LowPassFilter}
\text{parameter Real } T=1;
\text{Real } u, y(\text{start}=1);
\text{equation}
\text{T} \times \text{der}(y) + y = u;
\text{end LowPassFilter;}
\]

6.2.1  Records

The keyword records is a restricted form of class. To declare a class without equations you have to declare a record class.

\[
\text{record Filterdata}
\text{Real } T;
\text{end Filterdata}
\]

6.2.2  Packages

It is possible to have nested class declarations, which makes it possible to keep the maintenance of the name space for classes, i.e. to store a set of related classes within an enclosing class, so the classnames wouldn’t collide with each other. For this purpose a special kind of class, package, can be used. A package could consist of only declarations of constants and classes. To refer to the inner class, dot-notation is used.
6.2.3 Information Hiding

Having all components accessible from the outside is a bad principle, because hiding information is essential to keep the maintenance and not break the OO-principle about encapsulation. Hiding information is allowed in Modelica by using the heading `protected`.

protected Resistor;

Information hiding does not control interactive environments and for that reason it is possible to inspect protected variables. If a class B extends a class A then the protected variables in class A will be accessible in class B.

6.3 Connections

Using the keyword `connector` a restricted class which does not have equations can be used to define physical connections. For some mechanisms Kirschoff’s law i.e., that the currents of all wires connected at a node are summed to zero, is needed and similar laws apply to flows in a piping network and to forces and torques in mechanical systems. The default rule is that connected variables are set equal. Such variables are called across variables. Real variables that should be summed to zero are declared with prefix `flow`. Such variables are also known as through variables, in Modelica we assume that such variables are positive when the flow (or corresponding vector) is into the component.

```modelica
capital

connector Pin
  Voltage v;
  flow Current i;
end Pin

connect (Pin1, Pin2)
```

This connection connects the two pins so that they forms one node.

6.4 Partial Models and inheritance

A very important feature in order to build reusable classes is to define and reuse partial models. Since there are models which includes same types of components a class can be defined as a base for all these models using the keyword `partial model`. Using the keyword `extends` such a partial model can be extended or reused to build a complete model.

```modelica

partial model TwoPin
  Pin p, n;
  Voltage v;
equation
  v = p.v - n.v;
  p.i + n.i = 0;
end TwoPin
```
model Inductor
    extends TwoPin;
    parameter Real L (unit = 'H');
equation
    L * der(i) = v;
end Inductor

The feature is similar to inheritance in other languages. Modelica even supports multiple inheritance, i.e. several extends statements.

Modelica doesn’t use inheritance for classification and type checking. By inheriting all components of the base class, an extends clause can be used for creating a subtype relationship. But it is not the only way to create it. For example, if a class A is defined to be a subtype of class B and class A contains all the public components of B. B contains a subset of the components declared in A. This kind of subtype relationship is especially used for class parameterization.

### 6.5 Equations

Modelica is unlike most general purpose languages not primarily based on algorithms, but uses equations instead. For every model the programmer can define a number of equations describing the properties of the model. The equations defines the relation between the different quantities in the simulation.

The biggest reason why Modelica uses equation is that every simulation problem in fact is a mathematical problem. It also give the language a high abstraction level, because an equation is often more intuitive than a algorithm. When Modelica compiles a model (which may consist of many interconnected models) it will put all these equations together in one equation system, making it possible to solve any variable. Besides ordinary equations Modelica support more complicated equations, such as DAE.

### 6.6 Matrices

Besides solving equations using scalars it is also possible to use matrices and vectors. This makes it possible to define equations of matrices and let Modelica solve them. This is very powerful in many applications such as 3D-simulations. The usual algebraic operators +, -, *, / works in the same way for matrices and vectors as for scalars, but division is only defined with scalars as denominator. A few new functions exists that are used for vectors and matrices, such as cross(a,b) which calculates the cross product of two vectors.

To define a multidimensional variable (i.e. a vector or a matrix) the following syntax is used:

```plaintext
Real[3][3] i; // Defines a 3x3 matrix.
```

It is quite familiar for most programmers with experience from C or Java programming. To assign values to vectors and matrices, code like this might be used:

```plaintext
v = {1,1,1};
i = [1,0,0;
    0,1,0;
    0,0,1];
```
6.7 Repeated equations

In an equation block a sort of for-loop construct can be used. It does not operate in the same way as a loop construct in an imperative language. Instead it expands the equation in the loop to $n$ equations (where $n$ is the number of repeats in the loop). This is especially useful when using arrays. A small example that will assign the values of the faculty function to an array:

```plaintext
// we assume a to be declared as Integer[5] fac
a[1] = 1;
for i in 2:5 loop
  a[i] = a[i-1]*i;
end for;
```

The above loop will be expanded to something like this:

```plaintext
a[1] = 1;
a[2] = a[1]*2;
a[3] = a[2]*3;
a[4] = a[3]*4;
a[5] = a[4]*5;
```

6.8 Algorithms

Even though Modelica primarily uses equations for modeling it is also possible to use algorithms. An algorithm is usually contained in a specific class called a function. This is a major difference from most of the general purpose OOP languages where the usual relation is that a function is a part of a class, not a class in itself. A function class can not include a equation clause, but must have an algorithm clause instead. Inside an algorithm block imperative code is used. The syntax used look very much like Java or C with one big exception, the `:=` operator is used for assignments as in Pascal to distinguish an assignment from an equation which uses the `=` operator to indicate equality.

Modelica uses a very strict definition of functions. A function is not allowed to have any side effects at all. Therefore is it not possible for a function to have any internal (static) variables which are persistent between the calls to the function. This restriction also prohibit use of some internal functions which implicitly records state in the function, such as the \texttt{der()} function for the time derivate. The primary reason that this restriction exist is that without internal state, a function will always return the same result for a given input, and it is therefore possible to differentiate the function. It is possible for a function to return more than one variable. This is accomplished by the use of a tuple notation when performing a call to a function. The arguments to a function can be supplied in two ways, either the old fashioned way with ordered arguments as in C and Java or with the named parameter passing mechanism normally used in Modelica. A definition of a simple faculty function might look like this:

```plaintext
function fac
  input Integer a;
  output Integer f;
algorithm
  if(a <= 1)
    f=1;
```
else
    f = a * fac(a-1);
end fac;

It is also possible to define external functions which are implemented in some other language. This is a bit dangerous though, because it is up to the programmer to make the function safe and suitable for invocation from Modelica. An example of the definition:

function makeBSOD //A very morbid function
    input String message;
    output Integer status;
external
end makeBSOD;

6.9 Discontinuous models

Not all problems are of a continuous linear nature. Sometimes it is necessary to use different equations in different parts of the input set. For example, you may need to set a limit to a variable or use a special case of a formula for a certain value. This can be done by using the if-statement in the equation clause. For example, one can make the \texttt{abs()} function like this.

\[
y = \text{if } x < 0 \text{ then } x*(-1) \text{ else } x;
\]

The if-statement can also be used to make a component conditional, which means that it can operate in more than one way. This can be used to make it possible to disable parts of a model. This code demonstrates this feature.

model Resistor
    extends TwoPin.
    parameter Real R(unit="Ohm");
    parameter Boolean enabled;
    equation
        if enabled then i=u*r;
end Resistor

6.10 Discrete models

For some applications a linear time isn’t appropriate. Instead time needs to be discretized into intervals. Modelica provides some specific features for such systems. The when-statement is used to perform actions at certain events. These events can for example be generated by the built-in function \texttt{sample}. The sample function takes two arguments, one start time and one interval. An event is then generated every time the it is true that \( time = start + n \times interval \), where \( n > 0 \).

Output variables from a discrete block has to be declared with the modifier \texttt{discrete}, this will make the variable to hold its value between each sampling. Another specific built-in function is very usable in discrete models, the \texttt{pre()} function. This function will return the value of its argument (which is a variable) before the sampling. An example that will return the number of whole seconds passed since the start of the simulation:

model Seconds
    discrete output Real s = 0;
equation
    when sample(1,1) then
        s = pre(s) + 1;
end Seconds

6.11 Units and quantities

The most important datatype in Modelica is Real. When a variable of type Real is declared it is possible to set two attributes of the variable describing the variables unit and/or quantity. This information has no semantic meaning for the Modelica compiler or run-time environment, but it may be used by tools based on Modelica for presentation purposes. It may also be used by tools to implement control of units and quantities for equations. A variant of this is dimensional analysis, which is discussed in [7].

New variants of the Real type can be created in Modelica with the type keyword. This makes it possible to create new type which is bound to a specified unit and quantity. For example:

type Voltage = Real(final quantity="Voltage" unit="V");
7 Dymola

Dymola is a visual simulation tool with support for Modelica. It is fully integrated with a graphical editor, a compiler and a runtime environment. It was originally based on a proprietary simulation language, but supports nowadays Modelica as well. It is developed by the commercial company Dynasim, founded by Dr. Hilding Elmqvist, which also was one of the founder of the Modelica design group.

With this environment it is possible to make simulation by an intuitive drag and drop interface. There is a great number of library components included and each one is represented by a symbol. The symbols can then be connected, and complete simulations can be built without a line of manually written code. The source code for the components is available, making it easy to reuse the code for own components. It is also possible to combine components into new components, which then can be used in the same way as the existing library components.

The generated Modelica code can be compiled to C, which then can be compiled with a C compiler to an executable file. This file can then be executed in the runtime environment, where all the components parameters can be changed before running the simulation. All variables from the simulations components are recorded in discrete time intervals during simulation. This information makes it possible to plot graphs and in some cases show 3D animations of your simulation.

Dymola is the environment which is used for the following examples.

8 A simple example with Modelica

This first example will show a very simple circuit of electronic components. All the components are written from scratch, no models from the standard libraries are used. The circuit connects three resistors in parallel, and when the simulation is ran we can see what current flow through either one of them. This simulation is not very interesting because of its very static result, no variables varies with time. It is just a direct current source and a resistive load.

To be able to connect models in Modelica we have to define a so called connector, it is a class that describes the various quantities in that point. In electronics, a point has a voltage v and a flow of current i.

```modelica
connector OwnPin
    Real v(unit="V");
    flow Real i(unit="A");
end OwnPin;
```

The code above is the declaration of the connector class OwnPin. It is quite easy to understand, we define the class with its name and then we declare the variables. The variables are instance variables. According to Kirchoff’s first law, the sum of the currents to a node is zero. To make Modelica follow that rule we use the keyword flow, which makes the i variable ”sum-to-zero”.

The next step is to make a model for a resistor. In this case we choose to make an ideal resistor because it is very easy, its characteristics are very easy to describe with a simple equation. In this case, this equation will be Ohm’s law on its easiest form, \( U = I \times R \).

```modelica
model OwnResistor
    parameter Real R(unit="Ohm");
    OwnPin posPin;
    OwnPin negPin;
    Real voltage(unit="V");
```
Real curr(unit="A");
equation
  voltage = posPin.v - negPin.v;
  0 = negPin.i + posPin.i;
  curr = posPin.i;
  R*curr = voltage;
end OwnResistor;

After the definition follows the keyword parameter before the variable R. The parameter keyword allows the user to change the value of R after compilation. R is the value of the resistor in Ohms. The resistor also got two instances of the connector class OwnPin, which we defined earlier and variables for the current through the resistor and the voltage drop over it.

According to fundamental laws of electronic, the current in a serial connected circuit is the same at every point, therefore we can easily define the current through the resistor as the current in one of the connection points. This is done with the equation \( \text{curr} = \text{posPin.i} \). Now we just need to define the voltage drop over the resistor. To do that, we will need two equations. The voltage drop is the difference in voltage between the resistors two pins. This is defined by the equation \( \text{voltage} = \text{posPin.v} - \text{negPin.v} \). The voltage drop is also the result of an application of Ohm’s law for the resistor, \( u = i \times r \), here it will read \( R \times \text{curr} = \text{voltage} \).

The possibility to write equations like this makes Modelica very powerful and easy to use for people with only some knowledge of programming. It also decreases the complexity of the program.

One more component is needed to make a complete circuit, a power source. We define a DC source very easily like this:

model OwnDC
  parameter Real V(unit="V");
  OwnPin posPin;
  OwnPin negPin;
equation
  posPin.v = V;
  negPin.v = 0;
end OwnDC;

The definition is rather easy, we define the voltage V as a parameter so it is changeable after compilation. An ideal power source is very simple, it just has two connectors (OwnPin) with a difference in voltage between them. This difference is the voltage V. In this case the negative pin is set to always be zero, this is not a good solution, but it is good enough for this simple example.

To make this example complete, we just need to connect this models to each other. This is easy to do by just defining a new model, which got instances of all the other models we already made. This will be the main model which we will perform the simulation on.

model Ex1
  OwnResistor r1(R=100);
  OwnResistor r2(R=200);
  OwnResistor r3(R=400);
  OwnDC v1(V=100);
equation
  connect(v1.posPin, r1.posPin);
  connect(v1.posPin, r2.posPin);
  connect(v1.posPin, r3.posPin);
connect(v1.negPin, r1.negPin);
connect(v1.negPin, r2.negPin);
connect(v1.negPin, r3.negPin)
end Ex1;

The only new keyword in this model is connect. As parameters it takes the connectors which shall be interconnected. Here we connect every positive pin to each other and all the negative pins to each other. The circuit is now ready for compilation and simulation.

If we set the values of the three resistors to 100, 200 and 400 Ohms, the voltage to 100, run the simulation and then plots the graphs for the current through the resistors, we will end up with a graph like in figure 1.

![Figure 1: The current through the three resistors.](image)

We can check if this is a true result by calculating the composite resistance of the three resistors. This is done by the formula

\[
\frac{1}{(1/r1) + (1/r2) + (1/r3)}
\]

With the values above, the composite resistance will be about 57 Ohms. By Ohm’s law we calculate the current in the circuit, \( I = \frac{V}{R} \), here \( i = 100/57 \). If Modelica is right, this i will be equal to the sum of the current through either one of the three resistors.
9 Example 2 - Not quite as simple

The first example showed an extremely simple circuit with models made from scratch. This example will show a little more complicated example with homemade components. It will also demonstrate a few more advanced feature of Modelica, such as inheritance.

The circuit is very basic, an AC source, a resistor and a capacitor serial connected, see figure 2. From the previous example we reuse the OwnPin connector, because it does exactly what we want. But instead of using it directly in our new components we make a so called partial model for electric components with two pins. A partial model is a bit like a abstract class in other object oriented languages. It defines behavior but is not usable as it is, instead it need to be inherited before usage.

partial model OwnTwoPin
  OwnPin posPin;
  OwnPin negPin;
  Real voltage(unit="V");
  Real curr(unit="A");
equation
  voltage = posPin.v - negPin.v;
  0 = posPin.i + negPin.i;
  curr = posPin.i;
end OwnTwoPin;

If you compare the definition of the class above with the resistor in the first example, you will notice that some of the functionality in that class is implemented in the OwnTwoPin class. That part of the OwnResistor class is the part that is common for all electronic components with two pins. For example it defines the difference in voltage between the two pins as the voltage over the component.

We can now use this model to design new twopin components in a very simple way. After the last example, we are pretty familiar with the electronic properties of a resistor. Let us make a new resistor model using the OwnTwoPin partial model.

model NewOwnResistor
  extends OwnTwoPin;
  parameter Real R(unit="Ohm");
equation
  R*curr = voltage;
end NewOwnResistor;

The most interesting part here is the extends keyword, it means that the model reuse the parts in the OwnTwoPin model, often called inheritance. This simplifies the writing of this component, as we now only need one equation, Ohms law on its easiest form.

One of the strongest features with object orientation is code reuse, and this accomplished by the use of inheritance. To show that the code really is reused, we design a model for a capacitor (which will be a part of the circuit) based on the OwnTwoPin model.

model NewOwnCapacitor
  extends OwnTwoPin;
  parameter Real C(unit="F");
equation
  C*der(voltage) = curr;
end NewOwnCapacitor;
This code is rather straightforward, but if you look at the equation, you see a new function called `der()`. `der()` returns the value of its argument differentiated with respect to time.

The last model we use here is a an AC source. It's built like the other components with the help of OwnTwoPin partial model. Observe that we declared PI as a constant here. Usually this constant already exists in the standard math library, but we define it explicitly here just to show how constants are defined.

```model NewOwnAC
extends OwnTwoPin;
    parameter Real VA(unit="V") = 230;
    parameter Real f(unit="Hz") = 50;
    constant Real PI=3.14126;
    equation
        voltage = VA*sin(2*PI*time*f);
end NewOwnAC;
```

This models equation is rather simple too, we make use of trigonometric function sin() to generate the AC voltage. We use a built in variable time of which value is the time since the start of the experiment, and by multiplying this value with the desired frequency i Hz and $2 \times PI$ we get the angle for a certain time. The sinus value of that angle is then multiplied with VA, which is the peak voltage, to get the output voltage at every certain time.

We have also made a ground component, it is called OwnGround and is defined like this:

```model OwnGround
    OwnPin p;
    equation
        p.v = 0;
end OwnGround;
```

It is very simple. It consists of just one OwnPin, of which voltage is set to zero.

To make something useful, we need to connect these models to let them work together. We want to make a circuit like in figure 2.

This is accomplished by the following Modelica code:
model Ex2
    NewOwnResistor r1(R=1);
    NewOwnCapacitor c1(C=0.0001);
    NewOwnAC ac(f=10000);
    OwnGround g;
    equation
        connect(ac.posPin, c1.posPin);
        connect(c1.negPin, r1.posPin);
        connect(r1.negPin, g.p);
        connect(g.p, ac.negPin);
    end Ex2;

After compilation we need to set the parameters to the values in the circuit above. The time of the simulation need to be changed to just a fraction of a second to make the graph readable, because of the high frequency. Running the simulation for 1/1000 of a second yields result like in figure 3

![Graph](image)

Figure 3: The voltage over the capacitor and the current through the resistor.

With a little experience from electronics, we are not surprised over the results. This circuit is a capacitive load, and as such we can expect that the current and the voltage to be out of phase. This is exactly what we see in the graph above, where the voltage is approximately 90 degrees behind the current.
10 Conclusion

Modelica is a powerful multi-domain simulation tool. The object oriented approach of Modelica makes it easy to compose new models. The high level of abstraction, the use of equations and adherence to well known standards makes it easy to learn and use, even for engineers.

The most actual status of the Modelica project can be found at http://www.Modelica.org
References


A Glossary

Continuous time domain  The Continuous Time (CT) Domain aims to help the design of systems that have continuous dynamic, for example, analog circuits, mechanical systems, and the continuous environment for embedded systems. Models in the CT domain has the form of ordinary differential equations (ODEs).

DAE  Differential-Algebraic Equation. An example

\[ 0 = f(\dot{x}, x, y, u) \]

where \( x \) is the vector of unknowns that appear differentiated in the equation and \( y \) is the vector of unknowns that do not appear differentiated.

ODE  Ordinary Differential Equation. An example

\[ \frac{dx}{dt} = f(x, u, t) \]
\[ y = g(x, u, t) \]

where \( x \) is the state of the system, \( u \) is the input, and \( y \) is the output. Time \( t \) in the model is continuous, and \( \frac{dx}{dt} \) is the derivative of \( x \) with respect to time.

Polymorphism  The ability to substitute object of matching interface for one another at run-time [8].

B Questions

1. What is the purpose with Modelica?
2. Why is the Modelica language based on equations?
3. How is Modelicas abstraction obtained?