Modelica - A Unified Object-Oriented Language for System Modeling and Simulation

Peter Fritzson and Vadim Engelson

PELAB, Dept. of Computer and Information Science
Linköping University, S-58183, Linköping, Sweden
{petfr,vaden}@ida.liu.se

Abstract. A new language called Modelica for hierarchical physical modeling is developed through an international effort. Modelica 1.0 [http://www.Dynasim.se/Modelica] was announced in September 1997. It is an object-oriented language for modeling of physical systems for the purpose of efficient simulation. The language unifies and generalizes previous object-oriented modeling languages. Compared with the widespread simulation languages available today this language offers three important advances: 1) *non-causal* modeling based on differential and algebraic equations; 2) *multidomain* modeling capability, i.e. it is possible to combine electrical, mechanical, thermodynamic, hydraulic etc. model components within the same application model; 3) a general type system that unifies object-orientation, multiple inheritance, and templates within a single *class* construct.

A class in Modelica may contain variables (i.e. instances of other classes), equations and local class definitions. A function (method) can be regarded as a special case of local class without equations, but including an *algorithm* section.

The equation-based non-causal modeling makes Modelica classes more reusable than classes in ordinary object-oriented languages. The reason is that the class adapts itself to the data flow context where it is instantiated and connected. The multi-domain capability is partly based on a notion of *connectors*, i.e. certain class members that can act as interfaces (ports) when connecting instantiated objects. Connectors themselves are classes just like any other entity in Modelica. Simulation models can be developed using a graphical editor for connection diagrams. Connections are established just by drawing lines between objects picked from a class library.

The Modelica semantics is defined via translation of classes, instances and connections into a flat set of constants, variables and equations. Equations are sorted and converted to assignment statements when possible. Strongly connected sets of equations are solved by calling a symbolic and/or numeric solver. The generated C/C++ code is quite efficient.

In this paper we present the Modelica language with emphasis on its class construct and type system. A few short examples are given for illustration and compared with similar constructs in C++ and Java when this is relevant.

1 Introduction

1.1 Requirements for a modeling and simulation language

The use of computer simulation in industry is rapidly increasing. This is typically used

E. Jul (Ed.): ECOOP'98, LNCS 1445, pp. 67-90, 1998.

to optimize products and to reduce product development cost and time. Whereas in the past it was considered sufficient to simulate subsystems separately, the current trend is to simulate increasingly complex physical systems composed of subsystems from multiple domains such as mechanic, electric, hydraulic, thermodynamic, and control system components.

1.2 Background

Many commercial simulation software packages are available. The market is divided into distinct domains, such as packages based on block diagrams (block-oriented tools, such as SIMULINK[18], System Build, ACSL[19]), electronic programs (signal-oriented tools, such as SPICE[20], Saber), multibody systems (ADAMS[21], DADS, SIMPACK), and others. With very few exceptions, all simulation packages are strong only in one domain and are not capable of modeling components from other domains in a reasonable way. However, this is a prerequisite to be able to simulate modern products that integrate, e.g., electric, mechanic, hydraulic and control components. Techniques for general purpose physical modeling have been developed some decades ago, but did not receive much attention from the simulation market due to lacking computer power at that time.

To summarize, we currently have three following problems:

- High performance simulation of complex multi-domain systems is needed. Current widespread methods cannot cope with serious multi-domain modeling and simulation.
- Simulated systems are increasingly complex. Thus, system modeling has to be based primarily on combining reusable components. A better technology is needed in creating easy-to-use reusable components.
- It is hard to achieve truly reusable components in object-oriented programming and modeling

Disadvantage of block-oriented tools is the gap between the physical structure of some system and structure of corresponding model created by the tool. In block-oriented tools the model designer has to predict in advance in which way the equations will be used. It reduces reusability of model libraries and causes incompatibilities between blocks.

1.3 Proposed solution

The goal of the Modelica project[23] is to provide practically usable solutions to these problems, based on techniques for mathematical modeling of reusable components.

Several first generation object-oriented mathematical modeling languages and simulation systems (ObjectMath [11,13], Dymola [4], Omola [2], NMF [12], gPROMS [3], Allan [6], Smile [5] etc.) have been developed during the past few years. These languages were applied in areas such as robotics, vehicles, thermal power plants, nuclear power plants, airplane simulation, real-time simulation of gear boxes, etc.

Several applications have shown, that object-oriented modeling techniques is not only comparable to, but outperform special purpose tools on applications that are far beyond the capacity of established block-oriented simulation tools.

However, the situation of a number of different incompatible object-oriented modeling and simulation languages was not satisfactory. Therefore in the fall of 1996 a group of researchers (see Sect. 3.6) from universities and industry started work towards standardization and making this object-oriented modeling technology widely available.

The new language was called Modelica and designed for modeling dynamic behavior of engineering systems, intended to become a *de facto* standard.

Modelica is superior to current technology mainly for the following reasons:

- Object-oriented modeling. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- Non-causal modeling. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation.
- Physical modeling of multiple domains. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model becomes more natural in contrast to block-oriented modeling tools. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

1.4 Modelica view of object-orientation

Traditional object-oriented languages like C++, Java and Simula support programming with operations on state. The state of the program includes variable values and object data. Number of objects changes dynamically. Smalltalk view of object orientation is sending messages between (dynamically) created objects. The Modelica approach is different. The Modelica language emphasizes *structured* mathematical modeling and uses structural benefits of object-orientation. A Modelica model is primarily a declarative mathematical description, which allows analysis and equational reasoning. For these reasons, dynamic object creation at runtime is usually not interesting from a mathematical modeling point of view. Therefore, this is not supported by the Modelica language.

To compensate this missing feature *arrays* are provided by Modelica. An array is an indexed set of objects of equal type. The size of the set is determined once at runtime. This construct for example can be used to represent a set of similar rollers in a bearing, or a set of electrons around an atomic nucleus.

1.5 Object-Oriented Mathematical Modeling

Mathematical models used for analysis in scientific computing are inherently complex in the same way as other software. One way to handle this complexity is to use object-oriented techniques. Wegner [7] defines the basic terminology of object-oriented programming:

- *Objects* are collections of operations that share a state. These operations are often called *methods*. The state is represented by *instance variables*, which are accessible only to the operation's of the object.
- *Classes* are templates from which objects can be created.
- *Inheritance* allows us to reuse the operations of a class when defining new classes. A subclass inherits the operations of its parent class and can add new operations and instance variables.

Note that Wegner's strict requirement regarding data encapsulation is not fulfilled by object oriented languages like Simula or C++, where non-local access to instance variables is allowed.

More important, while Wegner's definitions are suitable for describing the notions of object-oriented *programming*, they are too restrictive for the case of object-oriented *mathematical modeling,* where a class description may consist of a set of equations, which implicitly define the behavior of some class of physical objects or the relationships between objects. Functions should be side-effect free and regarded as mathematical functions rather than operations. Explicit operations on state can be completely absent, but can be present. Also, causality, i.e. which variables are regarded as input, and which ones are regarded as output, is usually not defined by such an equation-based model.

There are usually many possible choices of causality, but one must be selected before a system of equations is solved. If a system of such equations is solved symbolically, the equations are transformed into a form where some (state) variables are explicitly defined in terms of other (state) variables. If the solution process is numeric, it will compute new state variables from old variable values, and thus operate on the state variables. Below we define the basic terminology of *object-oriented mathematical modeling*:

- An *object* is a collection of variables, equations, functions and other definitions related to a common abstraction and may share a state. Such operations are often called *methods*. The state is represented by *instance variables*.
- *Classes* are templates from which objects or subclasses can be created.
- *Inheritance* allows us to reuse the equations, functions and definitions of a class when defining objects and new classes. A subclass inherits the definitions of its parent class and can add new equations, functions, instance variables and other definitions.

As previously mentioned, the primary reason to introduce object-oriented techniques in

mathematical modeling is to reduce complexity. To explain these ideas we use some examples from the domain of electric circuits. When a mathematical description is designed, and it consists of hundreds of equations and formulae, for instance a model of a complex electrical system, structuring the model is highly advantageous.

2 A Modelica overview

Modelica programs are built from *classes*. Like in other object-oriented languages, class contains variables, i.e. class attributes representing data. The main difference compared with traditional object-oriented languages is that instead of functions (methods) we use *equations* to specify behavior. Equations can be written explicitly, like `a=b`, or be inherited from other classes. Equations can also be specified by the **`connect`** statement. The statement **`connect`**`(`*v1*`,` *v2*`)` expresses coupling between variables *v1* and *v2*. These variables are called *connectors* and belong to the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using Modelica.

In the following sections we introduce some basic and distinctive syntactical and semantic features of Modelica, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants. Powerful parametrization capabilities (which are advanced features of Modelica) are discussed in Sect. 2.10.

2.1 Modelica model of an electric circuit

As an introduction to Modelica we will present a model of a simple electrical circuit as shown in Fig. 1. The system can be broken 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 such components are available in Modelica class libraries.

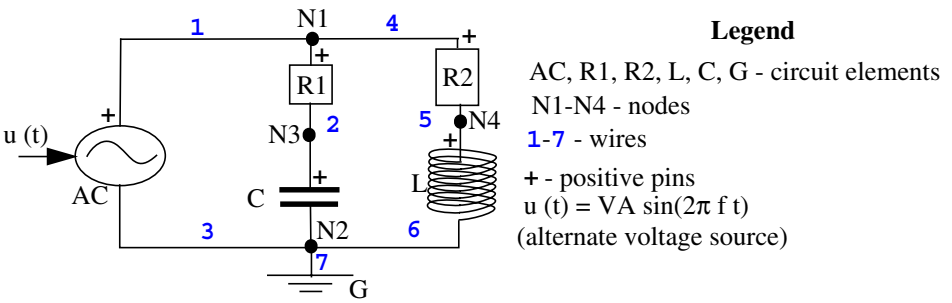

Fig. 1. A connection diagram of the simple electrical circuit example. The numbers of wires and nodes are used for reference in Table 3.1.

A declaration like one below specifies that `R1` to be of class `Resistor` and sets the default value of the resistance, `R`, to 10.

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

A Modelica description of the complete circuit appears as follows:

```
class 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);     // Wire 1
  connect (R1.n, C.p);      // Wire 2
  connect (C.n,  AC.n);      // Wire 3
  connect (R1.p, R2.p);     // Wire 4
  connect (R2.n, L.p);      // Wire 5
  connect (L.n,  C.n);      // Wire 6
  connect (AC.n, G.p);      // Wire 7
end circuit;
```

A composite model like the circuit model described above specifies the system topology, i.e. the components and the connections between the components. The connections specify interactions between the components. In some previous object-oriented modeling languages connectors are referred to cuts, ports or terminals. The keyword **connect** is a special operator that generates equations taking into account what kind of interaction is involved as explained in Sect. 2.3.

Variables declared within classes are public by default, if they are not preceded by the keyword **protected** which has the same semantics as in Java. Additional **public** or **protected** sections can appear within a class, preceded by the corresponding keyword.

2.2 Library classes

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

A connector must contain all quantities needed to describe an interaction. For electrical components we need the variables `voltage` and `current` to define interaction via a wire. The types to represent those can be declared as

```
class Voltage = Real;
class Current = Real;
```

where `Real` is the name of a predefined variable type. 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);
```

In Modelica, the basic structuring element is a **class**. There are seven restricted class categories with specific keywords, such as **type** (a class that is an extension of built-in classes, such as Real, or of other defined types) and **connector** (a class that does not have equations and can be used in connections). In any model the **type** and **connector** keywords can be replaced by the **class** keyword giving a semantically equivalent model. Other specific class categories are **model**, **package**, **function** and **record** of which **model** and **record** can be replaced by **class**[1].

The idea 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, generic properties 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 have 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. The previous definitions can be expressed as follows using the keyword **type** which is equivalent to **class**, but limits the defined type to be extension of a built-in type, record or array.

```
type Voltage = Real;
type Current = Real;
```

2.3 Connector classes

A connector class is defined as follows:

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

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 implies two equations[2],

---

1. The single syntax for functions, connectors and classes introduced by the **class** construct is a convenient way of notion unification. There is a similar approach in the BETA programming language[28] where classes and procedures are unified in the *pattern* concept.

2. The are other tools, for instance, PROLOG, where symbolic equations between terms can be written. However, in contrast to PROLOG, the Modelica environment automatically computes which variables of equations are inputs and which are outputs in corresponding context at the compilation phase. This leads to higher robustness and better performance.

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 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.

2.4 Virtual (partial) classes.

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

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` (Fig. 2).

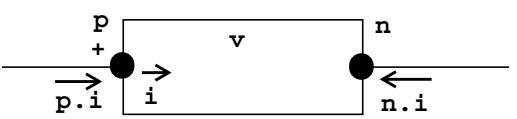

Fig. 2. Generic `TwoPin` model.

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 keyword 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. String after the class name is a comment.

2.5 Equations and non-causal modeling

Non-causal modeling means 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 equations-based models is unspecified and fixed only when the equation systems are solved. This is called non-causal modeling.

The main advantage with non-causal 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 telling which variables are needed as *outputs* and which are external *inputs* to the simulated system.

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

For example the equation from resistor class below:

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

can be used in two ways. The variable `v` can be computed as a function of `i`, or the variable `i` can be computed as a function of `v` as shown in the two assignment statements below:

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

In the same way the following equation from the class `TwoPin`

```
v = p.v - n.v
```

can be used in three ways:

```
v   := p.v - n.v;
p.v := v + n.v;
n.v := p.v - v;
```

2.6 Inheritance, parameters and constants

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

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

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

A Modelica **constant** never changes and can be substituted inline.

The keyword **extends** specifies the parent class. All variables, **equation**s and **connect**s are inherited from the parent. Multiple inheritance is supported in Modelica.

Just like in C++ variables, equations and connections of the parent class cannot be *removed* in the subclass.

In C++ a virtual function can be *replaced* by a function with the same name in the child class. In Modelica 1.0 the equations cannot be named and therefore in general equations cannot be replaced at inheritance[1]. When classes are inherited, equations are accumulated. This makes the equation-based semantics of the child classes consistent with the semantics of the parent class.

An innovation of Modelica is that type of a variable of the parent class can be replaced. We describe this in more detail in Sect. 2.10.

2.7 Time and model dynamics

Dynamic systems are models where behavior evolves as a function of *time*. We use a predefined variable `time` which steps forward during system simulation.

A class for the voltage source can be defined as:

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

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

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

1. In the ObjectMath language equations can be named and thus specialized through inheritance.

where **der**(v) means the time derivative of v.

During system simulation the variables i and v evolve as functions of time. The solver of differential equations (see Sect. 3.2) computes the values of *i(t)* and *v(t)* (*t* is time) so that *C v'(t)=i(t)* for all values of *t*.

Finally, we define the ground point as a reference value for the voltage levels

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

2.8 Functions

Sometimes Modelica non-causal models have to be complemented by traditional procedural constructs like function calls. This is the case if a computation is more conveniently expressed in an algorithmic or procedural way. For example when computing the value of a polynomial form where the number of elements is unknown, as in the formula below:

$$y = \sum_{i=1}^{size(a)} a_i \cdot x^i$$

Modelica allows a specialization of a class called **function**, which has only public inputs and outputs (these are marked in the code by keywords **input** and **output**), one **algorithm** section and no equations:

```
function PolynomialEvaluator
    input  Real a[:];// array, size defined at run time
    input  Real 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;
```

The Modelica function is side-effect free in the sense that it always returns the same outputs for the same input arguments. It can be invoked within expressions and equations,

e.g. as below:

```
p = PolynomialEvaluator2(a=[1, 2, 3, 4], x=time);
```

More details on other Modelica constructs are presented in [23].

2.9 The Modelica notion of subtypes

The notion of subtyping in Modelica is influenced by type theory of Abbadi and Cardelli [1]. The notion of inheritance in Modelica is separated from the notion of subtyping. According to the definition, a class `A` is a *subtype* of class `B` if class `A` contains all the public variables declared in the class `B`, and types of these variables are subtypes of types of corresponding variables in `B`. The main benefit of this definition is additional flexibility in the composition of types. For instance, the class `TempResistor` is a subtype of `Resistor`.

```
class TempResistor
  extends TwoPin
  parameter Real R, RT, Tref ;
  Real T;
equation
  v=i*(R+RT*(T-Tref));
end TempResistor
```

Subtyping is used for example in class *instantiation*, *redeclarations* and function calls. If variable `a` is of type `A`, and `A` is a subtype of `B`, then `a` can be initialized by a variable of type `B`. Redeclaration is discussed in the next section.

Note that `TempResistor` does not inherit the `Resistor` class. There are different equations for evaluation of `v`. If equations are inherited from `Resistor` then the set of equations will become inconsistent in `TempResistor`, since Modelica currently does not support named equations and replacement of equations. For example, the specialized equation below from `TempResistor`:

```
v=i*(R+RT*(T-Tref))
```

and the general equation from class `Resistor`

```
v=R*i
```

are inconsistent.

2.10 Class parametrization

A distinctive feature of object-oriented programming languages and environments is ability to fetch classes from standard libraries and reuse them for particular needs. Obviously, this should be done without modification of the library codes. The two main mechanisms that serve for this purpose are:

- *inheritance*. It is essentially "copying" class definition and adding more elements (variables, equations and functions) to it.
- *class parametrization* (also called generic classes or types). It is replacing a generic type identifier in whole class definition by an actual type[1].

In Modelica we propose a new way to control class parametrization. Assume that a library class is defined as

```
class SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

Assume that in our particular application we would like to reuse the definition of `SimpleCircuit`: we want to use the parameter values given for `R1.R` and `R2.R` and the circuit topology, but exchange `Resistor` with the temperature-dependent resistor model, `TempResistor`, discussed above.

This can be accomplished by redeclaring R1 and R2 as follows.

```
class RefinedSimpleCircuit = SimpleCircuit(
  redeclare TempResistor R1,
  redeclare TempResistor R2);
```

Since `TempResistor` is a subtype of `Resistor`, it is possible to replace the ideal resistor model. Values of the additional parameters of `TempResistor` can be added in the redeclaration:

```
redeclare TempResistor R1(RT=0.1, Tref=20.0)
```

This is a very strong modification but it should be noted that all equations that could be defined in `SimpleCircuit` are still valid.

2.10.1 Comparison with C++

The C++ language is chosen here as a common representative of an object-oriented language with a static type system. The reason to compare to C++ is to shed additional light on how the Modelica object model works in practice compared to traditional object-oriented languages. We consider the complications which arise if we attempt to reproduce Modelica class parametrization in C++.

A **`SimpleCircuit`** template class can be defined as a component of a C++ class library as follows:

---

1. The ObjectMath language[11] has class parametrization that allows replacing any identifier in whole class definition by an actual symbol.

```
class Resistor {
      public:
            float R;
};

template <class TResistor, class TResistor1>
   // Several template arguments can be given here
class SimpleCircuit {
  public:
   SimpleCircuit(){ R1.R=100.0; R2.R=200.0; R3.R=300.0; };
   TResistor  R1;//We should explicitly specify which two resistors will be
replaced.
   TResistor1 R2;
   Resistor R3;
   void func() {R3.R=R2.T;};};
```

Code which reuses the library classes should look like

```
class TempResistor {
      public:
            float R,T,Tref,RT;
};
class RefinedSimpleCircuit:public
   SimpleCircuit<TempResistor,TempResistor> {
   // Template parameters are passed
      RefinedSimpleCircuit(){ R1.RT=0.1; R1.Tref=20.0; }
...
};
```

To summarize we can reproduce the whole model in C++. However it is not possible to specify the `SimpleCircuit` class without explicitly specifying which data members (e.g. `R1` and `R2`) are controlled by a type parameter such as `TResistor`. The C++ template construct requires this. Therefore the possible use of type parameters in C++ always has to be anticipated by making types explicit parameters of templates. In Modelica this generality is always available by default. Therefore C++ classes typically are less general and have lower degree of reusability compared to Modelica classes.

2.10.2 Comparison with Java

Java is another object-oriented language with a static type system. There are no options for generic classes. Instead we can use explicit type casting. The same approach can be used in C++, using pointers. However, type casting gives clumsy and less readable code.

Example 1. If we permit `TempResistor` to be a subclass of `Resistor`, the code is straightforward:

```
class Resistor { public  double  R; };
class SimpleCircuit
{  public  SimpleCircuit() {
    R1=new Resistor();  R1.R=100.0;
    R2=new Resistor();  R2.R=200.0;
    R3=new Resistor();  R3.R=300.0;};
    Resistor R1, R2, R3;
    void  func(){R3.R=R1.R;};
 };

class TempResistor extends Resistor
{  public  double  T,Tref,RT; };
class RefinedSimpleCircuit extends SimpleCircuit
{ public
  RefinedSimpleCircuit() {
    R1=new TempResistor();  R2=new TempResistor();
   // Type casting is necessary below:

((TempResistor)R1).RT=0.1;((TempResistor)R1).TRef=20.0;}
};
```

There is no way to initialize and work further with the variables `R1` and `R2` without type casting.

Example 2: If we do not permit `TempResistor` to be a subclass of `Resistor`, the code is full with type casting operators:

```
class Resistor { public  double  R; };
class SimpleCircuit
{  public  SimpleCircuit() {
    R1=new Resistor();  ((Resistor)R1).R=100.0;
    R2=new Resistor();  ((Resistor)R2).R=200.0;
    R3=new Resistor();  ((Resistor)R3).R=300.0;};
    Object R1, R2, R3;
    void  func(){((Resistor)R3).R=((Resistor)R1).R;
    // This causes exception if R1 has runtime type TempResistor.
 };
};

class TempResistor
{  public  double  R,T,Tref,TR; };
class RefinedSimpleCircuit extends SimpleCircuit
{ public
  RefinedSimpleCircuit() {
    R1=new TempResistor(); R2=new TempResistor();
    // Type casting is necessary below
```

```
((TempResistor)R1).RT=0.1;((TempResistor)R1).TRef=20.0;}
};
```

The class `Object` is the only mechanism in Java that we can use for construction of generic classes. Since strong type control is enforced in Java, type cast operators are necessary for every access to `R1`, `R2` and `R3`. Actually we remove type control from compilation time into the run time. This should be discouraged because it makes the code design more difficult and makes the program error-prone.

To summarize we can reproduce the whole model in Java and build an almost general library. However, many explicit class casting operations make the code difficult and non-natural.

2.10.3 Final components

The modeler of the `SimpleCircuit` can state that a component cannot be redeclared anymore. We declare such component as final.

```
final Resistor R3(R=300);
```

It is possible to state that a parameter is frozen to a certain value, i.e. is not a parameter anymore:

```
Resistor R3(final R=300);
```

2.10.4 Replaceable classes

To use another resistor model in the class `SimpleCircuit`, we needed to know that there were two replaceable resistors and we needed to know their names. To avoid this problem and prepare for replacement of a set of classes, one can define a replaceable class, `ResistorModel`. The actual class that will later be used for `R1` and `R2` must have `Pins` `p` and `n` and a parameter `R` in order to be compatible with how `R1` and `R2` are used within `SimpleCircuit2`. The replaceable model `ResistorModel` is declared to be a `Resistor` model. This means that it will be enforced that the actual class will be a subtype of `Resistor`, i.e., have compatible connectors and parameters. Default for `ResistorModel`, i.e., when no actual redeclaration is made, is in this case `Resistor`. Note, that `R1` and `R2` are in this case of class `ResistorModel`.

```
class SimpleCircuit2
  replaceable class ResistorModel = Resistor;
protected
  ResistorModel R1(R=100), R2(R=200);
  final Resistor R3(final R=300);
equation
  connect(R1.p, R2.p);
```

```
  connect(R1.p, R3.p);
end SimpleCircuit2;
```

Binding an actual model `TempResistor` to the replaceable class `ResistorModel` is done as follows.

```
class RefinedSimpleCircuit2 =
  SimpleCircuit2(redeclare class ResistorModel =
TempResistor);
```

This construction is similar to the C++ **`template`** construct. `ResistorModel` can serve as a type parameter. However, in C++ the type parameter cannot have default value. In Modelica the class `SimpleCircuit2` is complete and can be used for variable instantiation. In C++ the class `SimpleCircuit2` is a template, which must be instantiated first:

```
template <class ResistorModel>
class SimpleCircuit2 {
    ResistorModel R1(R=100);
...
}
class RefinedSimpleCircuit2 : public
   SimpleCircuit2<TempResistor>
    { ... }
```

3 Implementation issues

3.1 Flattening of equations

Classes, instances and equations are translated into flat set of equations, constants and variables (see Table 3.1). As an example, we translate the `circuit` model from Sect. 2.1.

The equation `v=p.v-n.v` is defined by the class `TwoPin`. The `Resistor` class inherits the `TwoPin` class, including this equation. The `circuit` 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 set 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 `AC.p.v=R1.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 current law.

The complete set of equations generated from the `circuit` class (see Table 3.1) consists of 32 differential-algebraic equations. These include 32 variables, as well as `time` and several parameters and constants.

Table 3.1: Equations generated from the simple circuit model

AC	`0=AC.p.i+Ac.n.i`
	`AC.v=Ac.p.v-AC.n.v`
	`AC.i=AC.p.i`
	`AC.v=AC.VA*`
	`sin(2*PI*AC.f*time);`
		L	`0=L.p.i+L.n.i`
			`L.v=L.p.v-L.n.v`
			`L.i=L.p.i`
			`L.v = L.L*L.der(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*C.der(v)`
		flow at node	`0=AC.p.i+R1.p.i+R2.p.i //1`
			`0=C.n.i+G.i+AC.n.i+L.n.i //2`
			`0=R1.n.i+ C.p.i // 3`
			`0 =R2.n.i + L.p.i // 4`

3.2 Solution and simulation

After flattening, all the equations are sorted. Simplification algorithms can eliminate many of them. If two syntactically equivalent equations appear only one copy of the equations is kept.

Then they can be converted to assignment statements. If a strongly connected set of equations appears, these can be transformed by a symbolic solver. The symbolic solver performs a number of algebraic transformations to simplify the dependencies between the variables. It can also solve a system of differential equations if it has a symbolic solution. Finally, C/C++ code is generated, and it is linked with a numeric solver.

The initial values can be taken from the model definition. If necessary, the user specifies the parameter values (described in Section 2.6)Sect. 2.6. Numeric solvers for differential equations (such as LSODE, part of ODEPACK[14]) give the user possibility to ask about the value of specific variable in a specific time moment. As the result a function of time, e.g. `R2.v(t)` can be computed for a time interval $[t_0, t_1]$ and displayed as a graph or saved in a file. This data presentation is the final result of system simulation.

In most cases (but not always) the performance of generated simulation code (including the solver) is similar to hand-written C code. Sometimes Modelica is more efficient than straightforward written C code, because additional opportunities for symbolic optimization are used.

3.3 Current status

Language definition. As a result from 8 meetings in the period September 1996 - September 1997 the first full definition of Modelica 1.0 was announced in September 1997 [23, 29].

Work started in the continuous time domain, since there is a common mathematical framework in the form of differential-algebraic equations and there are several existing modeling languages based on similar ideas. There is also significant experience of using these languages in various applications. It is expected that the Modelica language will become a *de-facto* standard.

Translators. A translator from a subset of Modelica to Dymola is currently available from Dynasim [8]. We are currently developing another implementation of Modelica. This version of Modelica is integrated with Mathematica [9] similarly to ObjectMath [11,13]. This tool provides both the symbolic processing power of Mathematica, the modeling capability of Modelica, as well as the integrated documentation and programming environment offered by Mathematica notebooks. The tool also supports generation of C++ code [10] for high-performance computations.

Graphical model editors. A 2D graphical model editor can be used to define a model by drawing and editing an object diagram very similar to the circuit diagram shown in Fig. 1. Such a model editor is will be implemented for Modelica based on an similar existing editor for the Dymola language, called Dymodraw. The user can place icons that represent the components and connect those components by drawing lines between their iconic representations. For clarity, attributes of the Modelica definition for the graphical layout of the composition diagram (here: an electric circuit diagram) are not shown in examples. These attributes are usually contained in a Modelica model as annotations (which are largely ignored by the Modelica translator and used by graphical tools).

The ObjectMath inheritance graph editor exemplifies another kind of model editor, which supports both display and editing of the inheritance graph. Classes and instances are represented as graph nodes and inheritance relations are shown as links between nodes. Such an editor is also planned for Modelica. A partial port of the ObjectMath editor to Modelica currently exists.

Dynamic simulation visualization. There are currently exist at least two tools for dynamic visualization and animation of Modelica simulations. One of these tools is a 3D model viewer called Dymoview [8] for Dymola and Modelica, which displays dynamic model behavior based on simulations. It can produce realistic visualizations of mechanical models constructed from graphical objects such as boxes, spheres and cylinders. The visual appearance of a class must be defined together will class

definition. Simultaneously with animation any computed variables can be plotted. Another tool called MAGGIE is described in the next section.

Dynamic computational steering and animation environment. The MAGGIE (Mathematical Graphical Generated Interactive Environment) 3D model visualizer, is being developed by us. It supports automatic generation of visual appearance of objects, based on the structure of Modelica models.

Thus, a computational steering environment for the simulation application is automatically produced from the Modelica model. The user can interactively modify the structure of the automatically created 3D appearance of the model before start of visualization.

However, many parameters of the simulation and geometrical properties of the model can also be modified *during* run time, i.e. during simulation and animation. This provides immediate feedback to the user as changing model behavior and visualization. The user can specify the parameters in form-like dialog windows automatically generated based on the Modelica model. For this purpose we reuse our experience with generation of such graphical interfaces from C++ and ObjectMath data structures [24].

The simulation code, the interface for parameter control, and a graphical library based on the OpenGL toolkit are linked together into a single application which forms a computational steering environment. In contrast to traditional visualization tools this supports dynamically changing surfaces, that is useful in deformation modeling, hydrodynamics, and volume visualization. By linking all the software components into a single application we avoid saving large volumes of intermediate results on disk, which improves interactive response of our tool.

Initial experience with MAGGIE is described in [22, 25].

Checking tools. Additional information can be added to improve model consistency and reliability. One important example is the **`unit`** attribute, which provides more information in the user input forms as well as enabling more detailed consistency checking as a form of more detailed type checking. For instance, voltage can be defined as

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

Such a checker can prevent mixing of variables with incompatible units, for example below:

```
time + position
```

where both variables have type `Real` (which are compatible), but have different units – seconds and meters – which are incompatible.

Applications. Several recent papers discussing application of Modelica in different application domains have been written: gear box modeling[15], thermodynamics[16] and mechanics[17].

3.4 Future work

The work by the Modelica Committee on the further development of Modelica and tools will continue. Current issues include definition of Modelica standard libraries. The most recent meeting was in Passau in October 1997, and next one is planned in Linköping in January 1998. Several companies and research institutes are planning development of tools based on Modelica. In our case, we focus on an interactive implementation of Modelica integrated with Mathematica to provide integration of model design and coding, documentation, simulation and graphics.

Compilation of Modelica. Modelica compilers are developed by us, by Dynasim and planned by other partners. Parser, translator and symbolic solver are developed on the basis of the experience gained with the Dymola program to convert the model equations into an appropriate form for numerical solvers with target languages C, C++, and Java. Special emphasis will be made on using symbolic techniques for generating efficient code for large models.

Parallel simulation. Part of a parallel simulation framework for high performance computers earlier developed by PELAB and SKF for bearing applications [26] will be ported and adapted for use with Modelica in order to speed up simulation.

Experimentation environment. An integrated experimentation environment is currently being designed and implemented. This integrates simulation, design optimization, parallelisation, and support documentation in the form of interactive Mathematica notebooks including Modelica models, live graphics and typeset mathematics.

Library Development. Public domain Modelica component libraries are currently under development by the Modelica design group, for example by DLR in the mecatronics area, being used in gearbox applications [15]. Several extensions to these libraries are needed in order to model complex systems. These libraries will be available for use in education and in engineering books.

3.5 Conclusion

A new object-oriented language Modelica designed for physical modeling takes some distinctive features of object-oriented and simulation languages. It offers the user a tool for expressing non-causal relations in modeled systems. Modelica is able to support physically relevant and intuitive model construction in multiple application domains. Non-causal modeling based on equations instead of procedural statements enables adequate model design and high level of reusability.

There is a straightforward algorithm translating classes, instances and connections into a flat set of equations. A number of solvers have been attached to the generated code for computation of simulation results. The experience shows that Modelica is an adequate tool for design of middle scale physical simulation models.

3.6 Acknowledgments

The Modelica definition has been developed by the Eurosim Modelica technical committee under the leadership of Hilding Elmqvist (Dynasim AB, Lund, Sweden). This group includes Fabrice Boudaud (Gaz de France), Jan Broenink (University of Twente, The Netherlands), Dag Brück (Dynasim AB, Lund, Sweden), Thilo Ernst (GMD-FIRST, Berlin, Germany), Peter Fritzson (Linköping University, Sweden), Alexandre Jeandel (Gaz de France), Kaj Juslin (VTT, Finland), Matthias Klose (Technical University of Berlin, Germany), Sven Erik Mattsson (Department of Automatic Control, Lund Institute of Technology, Sweden), Martin Otter (DLR Oberpfaffenhofen, Germany), Per Sahlin (BrisData AB, Stockholm, Sweden), Peter Schwarz (Fraunhofer Institute for Integrated Circuits, Dresden, Germany), Hubertus Tummescheit (GMD FIRST, Berlin, Germany) and Hans Vangheluwe (Department for Applied Mathematics, Biometrics and Process Control, University of Gent, Belgium). The work by Linköping University has been supported by the Wallenberg foundation as part of the WITAS project.

References

[1] Abadi M., and L. Cardelli: *A Theory of Objects*. Springer Verlag, ISBN 0-387-94775-2, 1996.

[2] Andersson M.: *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD thesis ISRN LUTFD2/TFRT--1043--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994.

[3] Barton P.I., and C.C. Pantelides: *Modeling of combined discrete/continuous processes*. AIChE J., 40, pp. 966--979, 1994.

[4] Elmqvist H., D. Brück, and M. Otter: *Dymola --- User's Manual*. Dynasim AB, Research Park Ideon,Lund, Sweden, 1996.

[5] Ernst T., S. Jähnichen, and M. Klose: *The Architecture of the Smile/M Simulation Environment.* Proc.15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics, Vol. 6, Berlin, Germany, pp. 653-658, 1997

[6] Jeandel A., F. Boudaud., and E. Larivière: *ALLAN Simulation release 3.1 description* M.DéGIMA.GSA1887. GAZ DE FRANCE, DR, Saint Denis La plaine, FRANCE, 1997.

[7] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger,* 1 (1):9-87, August 1990.

[8] Dynasim Home page, `http://www.Dynasim.se`

[9] Mathematica Home page, `http://www.wolfram.com`

[10] Fritzson, P. Static and String Typing for Extended Mathematica, *Innovation in Mathematics*, Proceedings of the Second International Mathematica Symposium, Rovaniemi, Finland, 29 June - 4 July, V. Keränen, P. Mitic, A. Hietamäki (Ed.), pp 153-160.

[11] Peter Fritzson, Lars Viklund, Dag Fritzson, Johan Herber. High-Level Mathematical Modelling and Programming, *IEEE Software,* 12(4):77-87, July 1995.

[12] Sahlin P., A. Bring, and E.F. Sowell: *The Neutral Model Format for building simulation, Version 3.02.* Technical Report, Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.

[13] ObjectMath Home Page, `http://www.ida.liu.se/labs/pelab/omath`

[14] Hindmarsh, A.C., ODEPACK, A Systematized Collection of ODE Solvers, *Scientific Computing,* R. S. Stepleman et al. (eds.), North-Holland, Amsterdam, 1983 (Vol. 1 of IMACS Transactions on Scientific Computation), pp. 55-64, also `http://www.netlib.org/odepack/index.html`

[15] Otter M., C. Schlegel, and H. Elmqvist, Modeling and Real-time Simulation of an Automatic Gearbox using Modelica. In Proceedings of ESS'97 - European Simulation Symposium, Passau, Oct. 19-23, 1997.

[16] Tummescheit H., T. Ernst and M. Klose, Modelica and Smile - A Case Study Applying Object-Oriented Concepts to Multi-facet Modeling. In Proceedings of ESS'97 - European Simulation Symposium, Passau, Oct. 19-23, 1997.

[17] Broenink J.F., Bond-Graph Modeling in Modelica. In Proceedings of ESS'97 - European Simulation Symposium, Passau, Oct. 19-23, 1997.

[18] SIMULINK 2 - Dynamic System Simulation. `http://www.mathworks.com/ products/simulink/`

[19] ACSL software. `http://www.mga.com`

[20] Jan Van der Spiegel. SPICE - A Brief Overview. http://howard.engr.siu.edu/elec/faculty/etienne/spice.overview.html, http://www.seas.upenn.edu/~jan/spice/spice.overview.html

[21] ADAMS - virtual prototyping virtually anything that moves. `http://www.adams.com`

[22] V. Engelson, P. Fritzson, D. Fritzson. Generating Efficient 3D graphics animation code with OpenGL from object oriented models in Mathematica, In *Innovation in Mathematics. Proceedings of the Second International Mathematica Symposium,* Rovaniemi, Finland, 29 June - 4 July 1997, V.Keränen, P. Mitic, A. Hietamäki (Ed.), pp. 129 - 136

[23] Modelica Home Page `http://www.Dynasim.se/Modelica`

[24] V. Engelson, P. Fritzson, D. Fritzson. Automatic Generation of User Interfaces From Data Structure Specifications and Object-Oriented Application Models. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP96)*, Linz, Austria, 8-12 July 1996, vol. 1098 of Lecture Notes in Computer Science, Pierre Cointe (Ed.), pp. 114-141. Springer-Verlag, 1996

[25] V. Engelson, P. Fritzson, D. Fritzson. Using the Mathematica environment for

generating efficient 3D graphics. In *Proceedings of COMPUGRAPHICS/ EDUGRAPHICS*, Vilamoura, Portugal, 15-18 December 1997 (to appear).

[26] D. Fritzson, P. Nordling. Solving Ordinary Differential Equations on Parallel Computers Applied to Dynamic Rolling Bearing Simulation. In *Parallel Programming and Applications*, P. Fritzson, L. Finmo, eds., IOS Press, 1995

[27] SIMPACK Home page `http://www.cis.ufl.edu/mpack/~fishwick/simpack.html`

[28] M. Löfgren, J. Lindskov Knudsen, B. Magnusson, O. Lehrmann Madsen O*bject-Oriented Environments - The Mjølner Approach* ISBN 0-13-009291-6, Prentice Hall, 1994. See also Beta Home Page, `http://www.daimi.aau.dk/~beta/`

[29] H. Elmqvist, S. E. Mattsson: "Modelica - The Next Generation Modeling Language - An International Design Effort". In *Proceedings of First World Congress of System Simulation,* Singapore, September 1-3 1997.