C++ Toolbox for Object-Oriented Modeling and Dynamic Simulation of Physical Systems

Falko J. Wagner and Mikael Z. Poulsen
Department of Energy Engineering and Department of Mathematical Modeling
Technical University of Denmark

Abstract

This paper presents the efforts made in an ongoing project that exploits the advantages of using object-oriented methodologies for describing and simulating dynamical systems. The background for this work is a search for new and better ways to simulate physical systems.
Many physical systems can be decomposed into subsystems or components. In many ways these components can be regarded as objects. This leads to the idea of using object-orientation and aspects like inheritance and encapsulation to describe the hierarchy of subsystems or components.
Traditional approaches involve custom-built simulation programs using function-based programming languages like Fortran. Applications developed this way are not easily reused. The object-oriented concepts of decomposition and generalization support high reusability of the developed objects or components.
Further it is the aim of this project not only to use object-orientation on the level of component modeling, but to take it one step further, also making the simulation process object-oriented.
Together with the concepts developed in this project, a prototype toolbox in C++ is presented and examples of application within various physical areas are given. Using a standard programming language like C++ as the modeling and simulation environment makes it easy to design experiments and implement customized applications.

1 Introduction

This paper presents a concept for component based modeling and dynamic simulation of physical systems. As the systems being modeled get larger and more complex, so does the need for modeling and simulation tools that can cope with this growth.
One way to deal with this increasing complexity is to divide the system into smaller parts and focus the modeling effort on these, before connecting them again to form a model for the entire system.
This method is not new to the modeler. However the development of object-oriented modeling or, as in many cases, programming tools like Simula, Smalltalk, Eiffel and C++, makes this kind of modeling easier than before.
The focus of this paper is an object-oriented modeling and simulation tool that is implemented using an object-oriented programming language, in this case C++ [1]. C++ is not only used as the implementation language, but acts also as the modeling environment, i.e. the user interface.
The simulation tool supports non-causal modeling of many physical systems, and examples systems in the areas of electrics, mechanics and thermodynamics have been implemented and tested. Here the concept is illustrated by a simple mechanical system.
The benefits and drawbacks of this concept are outlined together with a discussion of the value of this kind of modeling with respect to interoperability and model understanding.

2 Object-oriented method

The object-oriented method combines a highly intuitive formalism for describing systems and subsystems with traditional modeling (and in many cases programming) techniques [2],[3] & [4].
The systems that are modeled are first analyzed and dissected into physically or conceptually distinct components. Objects are then used to

encapsulate the properties of these components. These properties are for mathematical modeling ODE's and DAE's together with look-up tables, design parameters and other numerical descriptors. For graphical representation of the model, also images and other kinds of visual representations can be stored in an object. An object is so to speak a smart container that can hold all the information necessary to model that particular component.

2.1 What is object-orientation?

It is a way of describing the world that helps you see things as individual objects rather than one entire undissectable system. It is a concept that is not very far from the way we look at things every day. We unavoidably see things as individual entities that have well defined boundaries and behavior. Especially if we see something, that we don't understand, we immediately ask ourselves "How does it work?" The understanding of things slowly grows, as we dissect them and try to understand parts of it at a time. When every part is understood, we put the parts together again and thereby gain understanding about the complete system.

Essential aspects of object-oriented modeling are:

- abstraction
- encapsulation
- decomposition
- inheritance

2.2 Abstraction

Using abstraction, the modeler only includes properties in the object or model, which are important for the purpose of the model. This is mainly done to keep the model relatively simple, to focus on what is essential and to help keeping the overview.

2.3 Encapsulation

This aspect enables the modeler to hide features or properties in private areas of the object. It is a way of implementing sides of a model, that should not be accessible for the user, but only for the object itself. This is done to ensure that nobody can interfere with this particular feature and induce inconsistency to the model in some sense.

2.4 Decomposition

Decomposition is used to split a model into more basic components. This is done to help the understanding of the model and to provide a greater possibility for reuse of model components. In object-oriented literature this concept is often called aggregation, which in fact is the opposite process, see also section 3.1

2.5 Inheritance

Closely related to decomposition, inheritance is a way to reuse general aspects of model components that are implemented in the decomposed parts. Through inheritance, one object or component can inherit the properties of another object.

3 Component-based modeling

Component-based modeling deals with decomposition of systems into smaller, well-defined components. It is a way of breaking down large and complex system into smaller – and thus more conceivable – parts focusing the modeling effort on these components.
This should be seen in contrast to a more function-oriented modeling, where the overview is easily lost as the systems grow in size.

3.1 Top-down-bottom-up

First, a system is decomposed into components. The individual components are modeled separately. In this step, the modeler can concentrate on modeling the individual component and its interaction with the surrounding environment in a more general way, without looking at the entire system. After modeling the individual components, aggregation can be used to form the complete system.

As one can see, this is closely related to the concept of object-orientation.

Physicists are sometimes in a peculiar situation, where they don't know how to dissect things further. Either they are not dissectable, or the physicists simply don't know how to dissect them. Considering the atom, for a long time, man thought, that this was the final dissection of everything. You could not split an atom. Hence the word "atom" from Greek "*atomos*", meaning "things, which are indivisible [6]". Later physicists have realized, that an atom consists of protons, neutrons and electrons, and called them elementary particles. That era did not last for long either, and now, protons and neutrons are found to be made out of quarks.

Why do they (the physicists) do that? To achieve a better understanding of the nature of all things!

The same method is used by the engineer to split a system into smaller parts, analyze them or model them, and then put them back together again. This is done to get a better representation and understanding of the system, than if he tried to model the whole system as one single item.
The object-oriented method provides a framework for this kind of modeling [7].

4 Example: The mechanical spring systems

The following example is a very simple example that is found useful to describe many aspects of component-based modeling of non-causal systems.

4.1 Decomposition

Consider the following textbook-example of a mechanical spring system (figure 1):

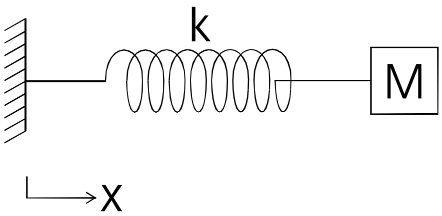

Figure 1 : The mechanical spring system

A spring, k, is attached to a fixed wall. Te other end of the spring is connected to a mass, M. The behavior of the system is well known, so it will not be elaborated.

The system is divided into conceptually distinct components. This gives us the following, (figure 2):

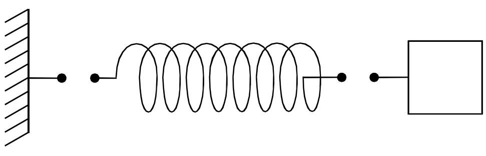

Figure 2 : Decomposition into components

Here the distinction is made between the wall, the spring and the mass as being individual components. Using OO terminology, they can be considered as individual objects (figure 3).

Information about the mathematical representation must be included in each object. Furthermore, the object must contain information concerning how it is interacting with other object / components.

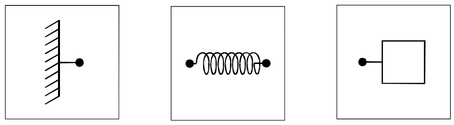

Figure 3 : Transformation of components into objects

The individual objects and their mathematical representation are described.

The wall:
A wall is an object that has a fixed position in space. As this example solely deals with one-dimensional motion, only one parameter for the wall is required. This is called X_0.

The spring:
The oscillating movement of the spring is described by the first order differential equation, combining forces, positions and velocities

$$F_1 + F_2 = -k(x_2 - x_1 - l) - \lambda(\dot{x}_2 - \dot{x}_1)$$

This is the only equation for the spring. Its parameters are the spring constant k, the spring length l, and the damping factor λ. x_1 and x_2 denote the positions of the ends of the spring, while
$\dot{x}_1$ and $\dot{x}_2$ are the respective velocities. F_1 and F_2 are the forces acting on the spring ends.

The mass:
Newton's second law describes the movement of the mass

$$m\dot{v} = F$$

Here the mass m is the only parameter, while F is the force acting on the mass and v is the velocity. This is a 2. order differential equation, since the correlation between position, velocity and acceleration is given by:

$$\dot{x} = v$$

In order to describe the mathematics of the mass,

there now are two equations for that component.
Hereby the individual components of the system are described to the level of abstraction, that was chosen for this example. Nevertheless, in order to describe the complete system, some additional equations have to be stated.

4.2 Aggregation

By connecting the wall, the spring and the mass, the remaining equations necessary to fully describe the system are provided. The connection between the wall and one end of the spring yields an equation, which sets the position of the wall equal to the position of the end of the spring. The other connection sets the position of the mass equal to the other end of the spring. Besides the position, the connections also take care of Newton's third law, stating that the forces acting on the components in the connection are equal in size, but with opposite direction.

This completes the description of the system.

5 Implementation in C++

This chapter deals with the object-oriented concept and prototype implementation of the toolbox in C++.
The reason why C++ was chosen for the implementation, is because it offers a very good object-oriented environment, and at the same time can produce highly efficient programs. Hereby the C++ programming environment becomes the implementation environment, and at the same time acts like the user interface. This offers possibilities both for component developers and for end users, who only want to build models from existing components and do simulations with them. It also makes it a very flexible basis for implementing new applications for and with the toolbox.

The following sections describe the design of the C++ classes, which are implemented in the toolbox and form the basis of the object-oriented modeling environment.

5.1 Components

First, the system is split into minor components. In the example above, the spring system was divided into a wall, the spring and a mass. These components will form the basis for the objects.
For describing the behavior of a component, equations are needed. Additionally the component needs terminals to connect it to other components. The condition of the component is described by its variables.

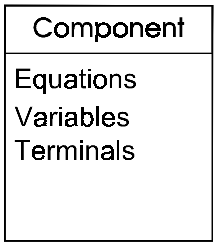

Figure 4 : The "Component" object

To provide hierarchical modeling capabilities, a component can contain other components. In this case, the component is called the supercomponent, whereas the components it contains are called subcomponents (see also figure 7).

5.2 Equations and expressions

Every equation consists of two expressions that are set equal, see figure 5. The equation can consequently be either true or false. In order to make it true, subtracting one expression from the other forms the residual. The residual, in its nature, has to be zero, which is achieved by the numerical routine (if possible!). As it is shown later the residual can be constructed in two ways.

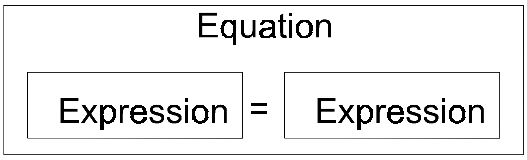

Figure 5 : The equation concept

Expressions can be anything from mathematical functions or flow profiles to vectors or simply a variable.

5.3 Terminals and connections

Terminals are the basis of the connections between the components. The idea is that through terminals or connections, elements in the components on either side are set equal. Typically, one variable in one component should be the same as another variable in the other component. Instead of variables, mathematical expressions can also be set equal through terminal connections (see figure 6). Variables are the simplest form of expressions.

Unlike other network based modeling tools, here a connection can only connect two components. In order to model multiple connection nodes (like in networks) a special component has to be declared, which takes care of the formalism behind this type of connections. This is done to ensure a general description of the individual components.

If three flows are connected in one connection, there is a modeling difference, whether the connection acts like a flow splitter or a flow junction. In the first case, the enthalpies of the two flows that flow away from the connection are automatically the same. In the second scenario, this is not the case, and an additional equation has to be applied. This modeling diversity has to be taken care of by a special component.

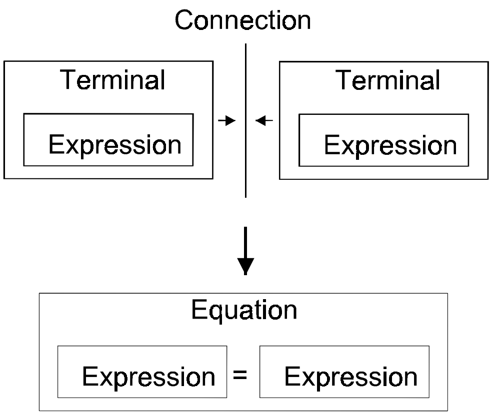

Figure 6 : Connections result in equations

When doing physical modeling there are two different ways, in which terminals can be connected, depending on the type of variable (or expression) they contain. In case of a simple variable, like a pressure, the values of the variable on either side must be identical. This is called an equality connection. But in case of a flow, like in the example above, the values of the variables on either side of the connection must be equal in size, but with opposite sign. This is because the flow is flowing out of one terminal, and into the other terminal. This type of connection is called a zero-sum connection.

5.4 The equation system

Once the model is completed and all components are connected, the equation system of the entire model is collected by retrieval of all equations from the components and from the connections. This equation system can then be passed on to the different numerical routines that can do different simulations with the system (see figure 7).

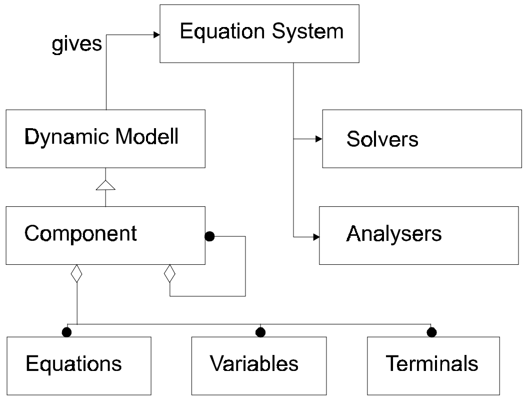

Figure 7 : Object hierarchy

5.5 The solvers

The concept provides the user with different solvers and analyzers. The model checker examines the system for consistency. It makes sure, that all components are connected properly, and that the equation system can possibly be solved with respect to number of variables and number of equations.
The static solver sets all derivatives to zero, looking for a static solution for the system. This is done using the least square method.
The state solver freezes all derivatives as they are set during initialization and uses the least square method to find a solution to the system in this particular state. This feature is useful when trying to find consistent initial conditions for the dynamic simulation.
The dynamic solver performs dynamic simulation with the system. It is build up around an implementation of the Godess system [5], which is a generic ODE solving system. The Godess system is implemented using object-oriented methodology, making it easy to incorporate in other object-oriented applications, such as this toolbox.
Godess is based on an idea, that should make it possible to implement different ODE and DAE solvers in one environment in such a way, that they are easily exchanged, making it possible to compare them on “paribus ceteris” conditions. It is consequently easy for the user to change the numerical routine to adapt the solver to different model types, without redesigning the interface to the solver.

5.6 Example

The example system from section 4 is used to demonstrate the implementation in C++.
The spring system consists of three components. These are implemented separately and then instantiated and connected to form the complete model.

Here it is seen that the model actually is a supercomponent (here called MultiComponent) and contains the `wall`, `spring` and `mass` components. The components are initialized with respective parameters.

```
class TModel : public TMultiComponent {
public:
   TWall wall;
   TSpring spring;
   TMass mass;

   TModel()
      : wall(0)
      , spring(1,1)
      , mass(1)

      RegisterSubComponent(wall);
      RegisterSubComponent(spring);
      RegisterSubComponent(mass);

      Connect(wall.term1,spring.term1);
      Connect(spring.term2,mass.term1);
   {};
};
```

Note the *registration* of the subcomponents, which is a side effect of using C++ aggregation. This was necessary to give the supercomponent "*knowledge*" of its subcomponents.

At last, the subcomponents are connected using the *connect* statement.
To illustrate the structure of a subcomponent, the mass component is shown below.

The mass is modeled with movement in one dimension, the position being `x`, the speed `v`. The force acting on the mass is `F`, and the mass is the parameter `m`. The residual functions of the two equations describing the movement of the mass are implemented as standard C functions and assigned to the two equations `Eqn1` and `Eqn2`. To connect to the spring, the mass has one terminal, denoted `term1`. A mechanical terminal is a terminal that consists of two other terminals. One is the force acting in the connection, the other is the position of the connection. The two subterminals `force` and `pos` are assigned to the variables `F` and `x` respectively.

Once the model class is implemented, it can be instantiated and used to perform different simulations. This is likewise done in an object-oriented fashion, since all the solvers are also objects.

```
class TMass : public TMultiComponent {
public:
   TMemberObjectDynamicVariable x,v,F;
   TClassMemberObjectEquation<TMass> Eqn1,Eqn2;
   TMechanicTerminal term1;
   TVariable m;

   double res1() const { return m*v.dot-F; };
   double res2() const { return x.dot-v; };

   TMass(TVariable m):m(m) {
      Eqn1.Set(this,&TMass::res1);
      Eqn2.Set(this,&TMass::res2);
      term1.force.Set(F);
      term1.pos.Set(x);

      RegisterEquation(Eqn1,Eqn2);
      RegisterDynamicVariable(x,v,F);
      RegisterTerminal(term1);
      F.Index=3;            // DAE index
      v.Index=2;
   };
};
```

The following examples shows a C++ main function, that instantiates two different solvers and the model from above, sets some parameters for the model and carries out the simulations.

```
void main() {
   TLSStateSolveFunction StateSolve;
   TGodessDynamicSolveFunction DynamicSolve;

   TModel TheModel;

   TheModel.SetTime(0);

   TheModel.mass.x=2;
   TheModel.mass.x.Independent();

   TheModel.mass.x.PrintAs("mass.x");
   TheModel.mass.v.PrintAs("mass.v");

   StateSolve(TheModel);

   DynamicSolve.TOL=1e-3;
   DynamicSolve.FileName='output.dat';
   DynamicSolve(TheModel,20);
};
```

The result of the simulation is well-known and is thus not repeated here.

6 Conclusion

Object-oriented modeling provides a good framework for component-based modeling. A prototype toolbox for object-oriented modeling of

non-causal physical systems has been implemented in C++.
By using C++ for the implementation, the user of the toolbox can benefit from a well-defined and well-known programming language in the implementation and model development phase, as well as in the end user or simulation phase.

The advantage from using an existing and well-known programming environment is that it provides object-orientation from the beginning. It also makes it easier to get started. On the other hand, C++ has been found to have some disadvantages too. There are modeling curiosities like the registration of subcomponents, and the fact that it is not so easy to understand for the novice.
Modeling tools like Modelica [8] are more user-friendly than C++, but on the other hand, the user has to learn a completely new modeling language.

Another feature, that is missing in C++ is the interactivity provided by modeling tools like Modelica or Dymola [9]. However, the user can benefit from the flexibility of the programming language.

As the main numerical solver, Godess was chosen. This was done mainly because Godess already existed as an object-oriented numerical solver, yet also to show the interoperability between different object-oriented tools, emphasizing aspects like genericity and reusability.

The implementation of an object-oriented modeling and simulation tool in C++ has proven to be a very beneficial combination.

7 References

[1] MIKAEL Z. POULSEN, FALKO J. WAGNER: *Objekt-orienteret modellering og dynamisk simulering.* Danmarks Tekniske Universitet, ET-ES 98-07, 2. udgave 1998

[2] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY & W. LORENSEN: *Object-Oriented Modeling and Design.* Prentice Hall, 1991

[3] E. YOUDON: *Object-Oriented Systems Design: An Integrated Approach.* Prentice Hall, 1994

[4] G. BOOCH: *Object-Oriented Analysis and Design, with Applications.* The Benjamin/ Cummings Publishing Company, Inc. 1994

[5] H. OLSSON: *Documentation of the structure of Godess.* Computer Science, Lund Institute of Technology, October 20, 1996

[6] ENCYCLOPÆDIA BRITANNICA ONLINE: *<http://search.eb.com/bol/topic?eu=119283& sctn=21&pm=1>*, Accessed 30 September 1999

[7] M. ABADI, L. CARDELLI: *A Theory of Objects – Monographs in Computer Science.* Springer Verlag, 1996

[8] H. ELMQVIST, S. E. MATTSSON, M. OTTER: *Modelica – an international effort to design an object-oriented modeling language.* Summer Computer Simulation Conference, 1998, Reno, Nevada, USA

[9] DYNASIM: *Dymola*, see http:\\www.dynasim.se\