Concepts Supporting Reuse of Models

Sven Erik Mattsson
Department of Automatic Control
Lund Institute of Technology
P.O. Box 118, S-221 00 LUND, Sweden

Abstract
Mathematical models of various kinds are important in many disciplines. Unfortunately, it is often difficult and time-consuming to develop models. This paper considers the possibilities to reduce model development work as far as possible by supporting reuse of models. Two basic ideas are discussed: First, use of symbolic descriptions (equations) to describe behaviour and second, concepts for modularization to make reuse of models both flexible and safe. The results presented are parts of project to develop computer aided engeneering tools for model development and simulation.

Introduction

Models of various kinds are important in all kinds of engineering. Since it is in general difficult and time-consuming to develop models, a basic question is "How can Computer Aided Engineering (CAE) tools support model development and simulation?"

Development of new models from scratch can be facilitated by proper CAE tools, but it will, as we can understand, always demand time and clever people. However, in many cases there are models available. Technical systems are often built from standard components for which there exist good models. Proper CAE tools can then support model development by making it easier to reuse models.

In this paper we will focus on the possibilities to support reuse of models. It should be possible to use a model for various tasks and it should be easy to modify a model to describe plants of similar types. The reuse of models should also be safe. The consistency of a model should be checked automatically.

Most of today's simulation languages and tools follow to their basic structure the CSSL definition (Strauss, 1967). The CSSL definition has had a profound impact on simulation. It has served well for over 20 years. During this time there has been an enormous development in computing science and computer hardware. In 1967 it was necessary to adapt the modeling languages to the computer. It is perhaps now time to reconsider the foundations of simulation and develop a modeling language more adapted to the user's need. To implement this, there are ideas to borrow from computer science.

We will discuss two complementary ideas. First, we will consider the basic concepts to describe behaviour. Second, we will discuss modularization of models to support reuse and building of model libraries. All concepts discussed are also useful when developing models from scratch.

Equations

Let us discuss what the model developer actually are defining when using today's simulation tools. When modeling heating, ventilation, and air conditioning (HVAC) systems, one uses fundamental laws as mass balances, energy balances and phenomenological equations. The natural mathematical notation is algebraic equations and differential equations. Unfortunately, most of today's simulation tools do not allow you to define your model as a differential/algebraic equation (DAE) system. You have to transform and translate the equations to a computational procedure.

Most of today's simulation tools for continuous time simulation solve problems of the form $\dot{x} = f(t, x)$ for given initial values, $x(t_0) = x_0$, where t is the time and $\dot{x}$ is the time derivative of x ($\dot{x} = dx/dt$). The model developer's task is to define a computational procedure which calculates the derivatives $\dot{x}$, when x and t are known. Most of today's tools allow also algebraic variables v. The user should for each derivative $\dot{x}_i$ and algebraic variable v_i define an assignment statement that defines its value. These assignment expressions may be given in any order as long as it possible to sort them without getting algebraic loops. It must be possible to sort them in such an order that every derivative $\dot{x}_i$ and every algebraic variable v_i have been assigned values before they are used in the right hand part of an assignment. Note that the states x_i are provided by the numerical integration algorithm

and therefore are known when calculating $f(t, x)$.

The requirement that the model developer should define a computational procedure for calculating the derivatives implies that he must transform his original equations. This is laborious. There is a risk of introducing errors. The documentation becomes bad, since it is difficult to recognize the original equations. There are also implications for the possibilities to use and build component libraries.

Let us as a typical example consider the facilities of HVACSIM$^+$ (Clark, 1985; Clark and May, 1985). A system model consists of a number of units connected together. The "unit" is the basic component and should be thought of as a model of a piece of equipment such as a pump, fan, pipe or heat exchanger. The connections between the units describe cause-and-effect relationships between inputs and outputs of the units. A connection is unidirectional saying that the value of an output should be calculated from the input connected. It means that the model developer must deduce the computational causality to define what are inputs and outputs to a unit. What are inputs and outputs of a unit are not only a property of the unit, but also of how it is used.

EXAMPLE 1—A simple pipe model
Let us consider a very simple pipe model, where the relation between the mass flow rate w from the inlet to the outlet, the pressure p_1 at the inlet and the pressure p_2 at the outlet of the pipe is given by $p_1 - p_2 = \text{sign}(w)Kw^2$. The flow resistance coefficient K should in the model be a parameter. The model has three variables p_1, p_2 and w. There are three possibilities to write the model on assignment form

$$p_1 := p_2 + \text{sign}(w)Kw^2$$
$$p_2 := p_1 - \text{sign}(w)Kw^2$$
$$w := \text{sign}(p_1 - p_2)\sqrt{|p_1 - p_2|/K}$$

The first two variants assume that the pressure at one end and the mass flow rate are inputs and calculate the pressure at the other end. HVACSIM$^+$ has such a pipe model called "Conduit (duct or pipe)". It is useful e.g. to calculate the inlet pressure needed to have a certain water flow out from the pipe into free air. The third variant assumes that p_1 and p_2 are inputs and defines w as an output of the model. HVACSIM$^+$ has also such a pipe model called "Inlet conduit (duct or pipe)". It is useful e.g. to model the flow from a water tank through a pipe out to free air. The water level in the tank defines the pressure at the inlet and the pressure at the outlet is the atmospheric pressure.

It is very inconvenient to need two mathematical equivalent variants of a pipe model. Unfortunately, the situation is worse. Assume that we want to model the flow from a water tank through two pipes A and B connected in series out to free air. No combination of the two pipe models can handle this case. There will always be an algebraic loop. An equation system involving variables of both pipes must be solved. □

The conclusion is that to support use of model libraries, the concepts for describing interactions should be equation based. The difficulty is that a tool supporting continuous simulation must be able to solve Differential/Algebraic Equation systems $g(t, \dot{x}, x) = 0$ which are more difficult to solve numerically than $\dot{x} = f(t, x)$. An overview of important properties of DAE systems can be found in Mattsson (1989a).

There are few tools supporting equation based behaviour descriptions. TRNSYS (1983), which is very similar to HVACSIM$^+$, can to some extent handle algebraic loops. In the context of building simulation there is the Simulation Problem Analysis Kernel, SPANK (Anderson, 1986; Sowell et al, 1986). It supports static calculations. Extensions to support dynamic simulation are discussed in Sowell and Buhl (1988). It may be remarked that the numerical routines to solve differential equations in TRNSYS, HVACSIM$^+$ and SPANK are poor. The approach is to use predictor-corrector methods to eliminate the derivatives and then solve a sequence of algebraic problem. Unfortunately, such an approach is too primitive for solving real problems. There is no automatic step length control, but the user has to select the step length himself. Furthermore, there is no estimation of the accuracy of the solution. There is no guarantee, that the result calculated is a solution to the problem. There are good numerical software like ODEPACK (Hindmarsh, 1983) that is free and easy to get as Fortran source code (Dongarra and Grosse, 1987).

Modularization

Modularization is a standard approach to beat complexity and to support reuse. Most simulation tools have some concepts for modularization of models.

In computing science much work has been focused on the possibilities to reuse software in different applications. The object oriented programming paradigm (Stefik and Bobrow, 1986) and the programming language Ada (Booch, 1983) are two examples. An important conclusion is that modules should be encapsulated with well-defined interfaces. The idea of data abstraction is the concept of separating the implementation details from the interface. Conversely, the internal details can be changed without affecting the way the module is used as a component.

The submodel concept

We propose a submodel concept, where a model consists of three parts: *terminals*, *parameters* and *realizations*. The terminals and the parameters constitute the interface of the model. Terminals are used to describe interaction with the environment. Parameters are used to adapt the description of behaviour. A realization is an internal description of the behaviour.

We distinguish between primitive realizations and structured realizations. A structured realization is decomposed into submodels and its behaviour is described by the submodels and their interaction. It means that the submodel concept is hierarchical. A primitive realization is not decomposed into submodels, but its behaviour is described in some mathematical or logical framework as differential equations, difference equations etc.

A model is allowed to have multiple realizations. Different realizations can describe the behaviour in more or less detail. It is an additional way to structure models. For example, one realization can describe the behaviour of the model under normal operation conditions and another realizations can describe the behaviour during exceptional conditions.

Component based model decomposition

The concepts proposed are general and allow model decomposition according to different principles, but we want to point out that a component based decomposition is a good alternative. It allows building of model libraries, so models for more complex systems can be built in a similar way as the system itself by composing existing submodels to make a new submodel. It facilitates reconfiguration of the model to investigate the effects of new or alternative components. It makes it natural for manufacturers of technical components to supply useful dynamic models instead of awkward data sheets.

Terminals

It is natural to aggregate terminal variables, since the description of an interaction between two subsystems often involves several quantities. With a hierarchical submodel concept it is natural to have a hierarchical terminal concept. At the lowest end we have simple terminals, which represent a quantity, which value can be defined and used in a primitive realization. We propose two types of composite terminals: record and vector terminals. Their subterminals can be simple, record or vector terminals. For a record terminal the subterminals can be of different types while a vector terminal accepts only one type of subterminals.

It is important to have standards and common guidelines for selection of terminals to make it possible to put together submodels developed at different places without recoding and adapters. If the submodel concept is component based, it is reasonable to anticipate the various ways a component could be connected to other components. Libraries of terminals can be built within certain application domains or companies to give guidelines and to standardize the selection of terminals.

EXAMPLE 2—Mass flows
The media entering or leaving the end of a pipe. can be described by a record terminal. The subterminals can be simple terminals describing the flow rate, pressure, temperature, enthalpy etc and a vector terminal describing the composition. □

Connections

Interactions between submodels of a structured realization are described by terminal connections. The term "connection" reflects what we are doing in the block diagram when describing an interaction as well what we are doing when building a physical plant by putting components together.

A connection between two structured terminals means that their first components should be connected to each other and so on recursively down to the level of simple terminals. It is useful to have two sorts of simple terminals: across and through. A connection between two across terminals mean that they are equal. Examples of physical quantities are position, pressure, temperature and voltage. Through terminals have an associated direction (in or out) and connected terminals should sum to zero. Examples of through quantities are mass flow, energy flow, force, torque and current.

Hibliz

The decomposition of a model as connected submodels is more easily described graphically than textually. A block diagram is a good way of describing model structure. We have developed a prototype simulator called Hibliz (Elmqvist and Mattsson, 1989), which exploits some features of modern computer graphics. The model developer can describe the model structure by drawing a block diagram in a Mcintosh like way. By moving the mouse and pressing its buttons, the user can scroll, pan and zoom the block diagram continuously in real time. As an example Figure 1 shows a model of a thermal power plant. The annotated boxes represent submodels, and the lines between the boxes are connections between terminals. When zooming in on a block, it changes to a representation showing the internal structure with increasing detail, information zooming (Elmqvist, 1985). See Figures 2 and 3.

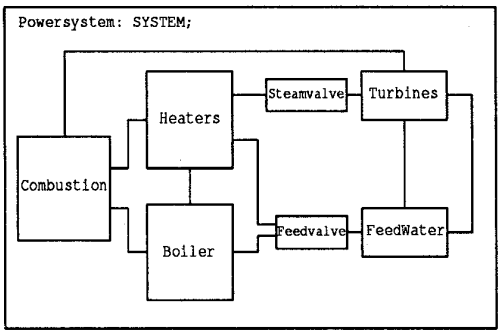

Figure 1. A power system model.

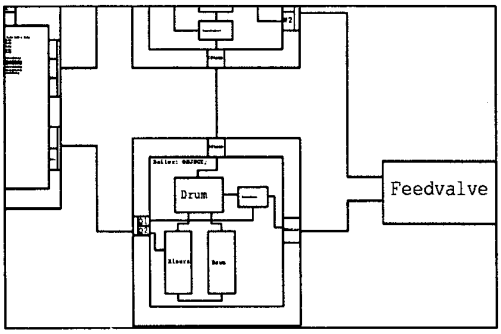

Figure 2. The model zoomed-in a bit.

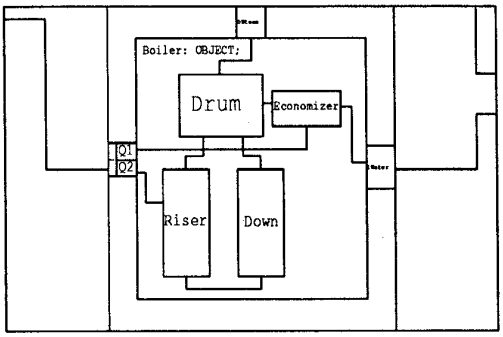

Figure 3. The model zoomed-in at Boiler.

Model types

In a complex system, a component may appear several times. For example a heating system may have several radiators. The maintenance of a model for this system is facilitated if the models for all those radiators share a common description. Common descriptions are supported by a model type concept. The typical radiator is described as a model and every occurrence of a radiator in the heating system references to this very same radiator model. When a model is to be simulated, every occurrence must have its own unique variables, but this instantiation can be done automatically without the user's concern.

An important idea in object oriented programming is inheritance and programming through specialization. It means that it is possible to define model types as specializations of previously defined and more general model types. Use of object oriented programming ideas in continuous time simulation are discussed by Andersson (1989).

Comparison with today's tools

The model type concept is usually supported by today's simulation tools. Model decomposition is typically handled by macros and the model structure is described textually. The "terminals" must be scalar and inputs or outputs.

The drawback with the macro concept is that the model structure is not preserved at compilation. The macros are expanded at compilation and at simulation the model has no structure. The names of the internal states and the variables of the submodels are replaced by names like `QQQQ1`, `QQQQ2` etc that are generated automatically to resolve potential name conflicts.

We think that the user should be allowed to inspect internal details at simulation. If he is simulating the heating system of a big house, he should be allowed to display the inlet temperature of the radiator in the kitchen of department 3 on the sixth floor. He should be able to reference it using a notation similar to that used in the preceding sentence or by support of hierarchical menus. He should not be forced to examine cross reference lists to find out that the name of the variable is `QQQQ153`.

Object oriented programming has its roots in discrete event simulation, but is not supported in continuous simulation today.

Consistency of connections

An important task is to make the use of library models safe and reliable. The encapsulation of models prevents to a large extent unintended abuse, but the terminals are dangerous holes in the wall. It would be nice if the user could get automatic warnings when making improper connections. To allow automatic consistency checks, the model developer must "supply" redundant information.

Simple terminals have the attributes name of quantity, unit of measure and value range. The name of quantity is used used to check the consistency of connections. There is an international standard (ISO 31)

for naming of quantities in different national languages (English, German, French, Swedish, Danish etc). We have used this standard to implement a database, which can answer questions about compatibilities of quantities and knows among other things the SI unit of each quantity. HVACSIM$^+$ supports to some extent this kind of consistency check, since inputs and outputs must belong to one of eight categories and it is only possible to connect inputs and outputs belonging to the same category. The categories are: pressure, mass flow rate, temperature, control signal, rotational speed, energy, power and absolute humidity.

The information about unit of measure is used for automatic introduction of proper scale factors in the connection equations, thus eliminating the need of user defined adapters. The value of the unit of measure attribute may be any unit expression. It must be consistent with the quantity attribute.

Information about ranges of validity are used to test for unintended abuse during simulations.

Unspecified terminal attributes

A declaration of a terminal may leave attributes unspecified. It is not our aim to force a user who, for example, is in an exploratory phase, to specify things that the computer itself can deduce from the context. The requirement that all connections should be consistent can be used to deduce unspecified properties. A model developer is hopefully better motivated to supply redundant information when he has tested the model and is going to include it in a public model library. Such an information can be seen as a part of the model documentation. Another case when it is useful to be allowed to give incomplete specifications is when making models which in some aspects are generic. Use of unspecified terminal attributes are discussed further in Mattsson (1989b).

Parameters

Parameterization is a way of introducing flexibility. A parameter is a time invariant variable that can be set from outside to modify a realization. The burden of a user to set parameters can be relieved by letting the model developer provide default values. If a good default alternative is provided, the casual user could be left unaware about the flexibility and no extra burden is put on him.

It is possible to define relations between parameters in the same way as between time varying variables. Relations between parameters have several applications. When a number of components are put together to form a new and more complex component, the components usually must fulfil some constraints to fit together and to fulfil the specifications. For example a simple connection of two pipes may demand that the shapes of the two cross sections are equal. Second, a model developer may want to assume certain relations between the parameters of different submodels to make the new model less complex. For example a model may assume that a number of components of the same type have identical parameter values. Third, a model may be parameterized in several ways. Different sets of physical parameters can be used. On higher levels it is natural to introduce performance and functional related parameters instead of parameters with a direct physical interpretation.

A terminal component may be a parameter. A terminal describing the end of a pipe can have a component describing the diameter of the cross section. There are two possible complementary uses of terminals which are defined to be parameters. First, they can be used for consistency checks to check if a connection is permitted and if the parameter settings in different submodels are consistent. When connecting two pipes, it is automatically checked that they have the same diameters. Second, they can be used to propagate parameter values from one submodel to another automatically. For example, it is convenient to be able to write a generic pipe model which pipe diameter is unspecified. The name of the medium can be used as an index parameter to retrieve physical and chemical parameters of the medium.

Conclusions

To support reuse of models for various purposes and building of model libraries, the model languages should be declarative, describing relations and properties. We believe that various users could agree upon a small, basic, common set of concepts; A collection of basic objects as *models*, *terminals*, *parameters*, *realisations* and *variables* with specified properties and operations. These basic objects can have various textual and graphical representations to support customized user interfaces. To allows various user interfaces and introduction of new tools, it is important to separate the user interface, the processing tools and the database properly. The design of the model representation is important, since it is common to all tools.

The basic concepts and tools are mainly intended for researchers and modeling and simulation specialists. Other user categories can be supported by building new user interfaces and new layers of tools. Such tools can allow an architect to describe his building and the assumptions in his own language. The tools should then generate the desired model in an explicit form as outlined above. It means that the generated

model is readable and can be modified by the user. Today's "high-level" tools of this kind like DOE-2 (1980) are too rigid. They produce canned, black box models which cannot be modified. The user is in trouble if some component is missing, since it is very difficult or even impossible for him to add new components.

The results reported in this paper are parts of a project to develop tools for model development and simulation (Mattsson, 1988, 1989b; Andersson, 1989). Our design includes also an internal representation to support these concepts. Common Lisp and KEE*. The advantage of KEE is that we have been able to develop prototypes with a small programming effort. But KEE is expensive and requires powerful workstations. We are investigating the possibilities to make an implementation that can be made generally available.

Acknowledgements

The author would like to thank Professor Karl Johan Åström, Mats Andersson, Bernt Nilsson and Dag Brück for many useful discussions. This work has been supported by the National Swedish Board for Technical Development (STU) under contract 87-2503.

References

ANDERSON, J.L. (1986): "A Network Definition and Solution of Simulation Problems," LBL-21522, Simulation Research Group, Applied Science Division, Lawrence Berkeley Lab., Berkeley, Ca.

ANDERSSON, M. (1989): "An Object-Oriented Modeling Environment," *Proc. of the 1989 European Simulation Multiconference*, Rome, June 7–9, 1989.

BOOCH, G. (1983): *Software Engineering with Ada*, The Benjamin/Cummings Publ. Comp.

CLARK, D.R. (1985): "HVACSIM+ Building Systems and Equipment Simulation Program – Reference Manual," NBSIR 85-3243, U.S. Department of Commerce, National Bureau of Standards, Gaithersburg, Maryland.

CLARK, D.R. and W.B. MAY, JR (1985): "HVACSIM+ Building Systems and Equipment Simulation Program – Users Guide," NBSIR 85-3243, U.S. Department of Commerce, National Bureau of Standards, Gaithersburg, Maryland.

* Knowledge Engineering Environment, KEE is a trademark of IntelliCorp, Inc.

DOE-2 (1980): "DOE-2 User's Guide, Version 2.1," Building Energy Analysis Group, Lawrence Berkeley Laboratory, Berkeley, California.

DONGARRA, J.J. and E. GROSSE (1987): "Distribution of Mathematical Software Via Electronic Mail," *Comm. ACM*, **30**, 403–407.

ELMQVIST, H. (1985): "LICS – Language for Implementation of Control Systems," CODEN: LUTFD2/TFRT-3179, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. and S.E. MATTSSON (1989): "Simulator for Dynamical Systems Using Graphics and Equations for Modeling," *IEEE Control Systems Magazine*, **9**, 1, 53–58.

HINDMARSH, A.C. (1983): "ODEPACK, a systemized collection of ODE solvers," *Scientific computing*, Stapleman et. al. (Eds.), North-Holland, pp. 55–64.

MATTSSON, S.E. (1988): "On Model Structuring Concepts," *Preprints of the 4th IFAC Symposium on Computer-Aided Design in Control Systems (CADCS)*, August 23–25 1988, P.R. China, pp. 269–274.

MATTSSON, S.E. (1989a): "On Modelling and Differential/Algebraic Systems," *Simulation*, **52**, No. 1, 24–32.

MATTSSON, S.E. (1989b): "Modeling of Interactions between Submodels," *Proc. of the 1989 European Simulation Multiconference*, Rome, June 7–9, 1989.

SOWELL, E.F., W.F. BUHL, A.E. ERDEM and F.C. WINKELMANN (1986): "A Prototype Object-Based System for HVAC Simulation," LBL-21522, Simulation Research Group, Applied Science Division, Lawrence Berkeley Lab., Berkeley, Ca.

SOWELL, E.F. and W.F. BUHL (1988): "Dynamic Extension of the Simulation Problem Analysis Kernel (SPANK)," Simulation Research Group, Applied Science Division, Lawrence Berkeley Lab., University of California, Berkeley, Ca.

STEFIK, M. and D.G. BOBROW (1986): "Object-Oriented Programming: Themes and variations," *AI Magazine*, **6:4**, 40–62.

STRAUSS, J.C. (Ed.) (1967): "The SCi Continuous System Simulation Language (CSSL)," *Simulation*, Dec 1967, 281–303.

TRNSYS (1983): "A Transient Simulation Program," Solar Energy Laboratory, University of Wisconsin, Wisconsin, USA.