A Kernel for System Representation

Mattsson, Sven Erik; Andersson, Mats

1989

Document Version:
Publisher's PDF, also known as Version of record

CODEN:LUTFD2/(TFRT-7429)/1-006/(1989)

A Kernel for System Representation

Sven Erik Mattsson
Mats Andersson

Department of Automatic Control
Lund Institute of Technology
August 1989

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
August 1989

Document Number
CODEN:LUTFD2/(TFRT-7429)/1-006/(1989)

Author(s)
Sven Erik Mattsson
Mats Andersson

Supervisor

Sponsoring organisation
The Swedish Board of Technical Development

Title and subtitle
A Kernel for System Representation

Abstract

This paper proposes a kernel for model representation. The kernel may serve as a central model data base in an integrated environment for model development and simulation. The CSSL definition from 1967 has had a profound impact on simulation and has served very well for over 20 years. It is perhaps now time to capitalize on the enormous development of information technology and reconsider the foundations of model representation. This paper is a modest effort in this direction. If we could agree upon a common set of ideas we may lay the foundation to a new standard. The proposed kernel supports a modularized and object oriented representation of models to allow flexible and safe reuse of model components. The model developer may supply extra information which is used for automatic consistency analysis to check for unintended abuse of models. The kernel can allow any logical and mathematical framework such as differential-algebraic equations, difference equations, etc. to describe behaviour, but a basic idea is that behaviour descriptions should be declarative and equation based. The kernel allows integration of different customized user interfaces. A prototype of the kernel is implemented in Common Lisp and KEE. An implementation in C++ is under development.

Key words
Computer aided system design; modeling; simulation languages; hierarchical systems; data structures.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
6

Recipient's notes

Security classification

11th IFAC World Congress, 13-17 August 1990, Tallin, USSR

A Kernel for System Representation

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

Abstract. This paper proposes a kernel for model representation. The kernel may serve as a central model data base in an integrated environment for model development and simulation. The CSSL definition from 1967 has had a profound impact on simulation and has served very well for over 20 years. It is perhaps now time to capitalize on the enormous development of information technology and reconsider the foundations of model representation. This paper is a modest effort in this direction. If we could agree upon a common set of ideas we may lay the foundation to a new standard. The proposed kernel supports a modularized and object oriented representation of models to allow flexible and safe reuse of model components. The model developer may supply extra information which is used for automatic consistency analysis to check for unintended abuse of models. The kernel can allow any logical and mathematical framework such as differential-algebraic equations, difference equations, etc. to describe behaviour, but a basic idea is that behaviour descriptions should be declarative and equation based. The kernel allows integration of different customized user interfaces. A prototype of the kernel is implemented in Common Lisp and KEE. An implementation in C++ is under development.

Keywords: Computer aided system design; modeling; simulation languages; hierarchical systems; data structures.

Introduction

Mathematical models of various kinds are important in all kinds of engineering. Modelling is often a time consuming part of an engineering job but it can be facilitated by proper Computer Aided Control Engineering (CACE) tools.

Today's most used languages for continuous simulation (ACSL, CSMP, CSSL IV, EASY5 etc., for overviews see Kreutzer (1986) and Kheir (1988)) follow the CSSL definition (Strauss, 1967). It has served very well for over 20 years. During this time there has been an enormous development of computing science and computer hardware. In 1967 it was necessary to adapt the modelling language to the computer. It is perhaps now time to reconsider the foundations of model representation and borrow ideas from computer science and develop a representation more adapted to the user's need. This paper is a modest effort in this direction. If we could agree upon a common set of ideas we may lay the foundation to a new standard.

This paper proposes a kernel for system representation. A kernel can be viewed as a central model data base in an integrated environment for model development, simulation and design. It is important that a kernel of this kind supports a flexible framework for representing many various aspects of model structure and behaviour. The aim is to support a common representation of models which allows flexible and safe reuse of models for various tasks and for similar systems. It i also important that the kernel representation is separated from the user interface, so that customized interfaces for various needs and user categories can be supported.

The paper is organized as follows. First,

the importance of a declarative equation based model description is discussed. Then the basic model structuring concepts are presented. Thirdly, we show how these concepts map into concepts in object-oriented programming.

Declarative models

Today models developed to be used in one CACSD package cannot without additional work be used in another. Unfortunately, much "model development" work of today consists of manual recoding or implementation of adapters. An obvious reason is of course that there is no common agreement on the representation of models.

Another maybe less obvious reason is that the representations used in most of today's CACSD and simulation tools are too specialized and of too low a level to allow reuse of models for other tasks than simulation. The CSSL tools solve problems of the type $dx/dt = f(t, x)$ if the user defines a Fortran-like procedure which calculates $f(t, x)$.

To allow a model to be used for different purposes it should be declarative and not procedural. It should describe facts and relations and not be a calculation procedure. A natural declarative form for continuous time models are Differential-Algebraic Equation (DAE) systems, $g(t, \dot{x}, x) = 0$. An overview of important properties can be found in Mattsson (1989a).

A declarative model is multipurpose, since it is symbolic and can be manipulated automatically to generate efficient code for simulation, code for calculation of stationary points, linear representations, descriptions which are accepted by other existing packages etc. Models of the controller can be used for automatic generation of the control software or to generate layouts for special purpose analog or digital VLSI circuits which implement the controller.

A declarative model is usually also closer to the modeller's perception of the physical reality, and therefore, development of new models is easier.

Model structures

To understand large models and to be able to reuse parts of models, good structuring facilities must be supported. A powerful modularization concept supports model development by beating complexity as well as it allows reuse of parts and building of models by putting together existing components.

An important conclusion from computer science is that modules should be encapsulated with well-defined interfaces. The idea is to support abstraction by separating the internal details of a model from its interface. It means also that internal details can be changed without affecting the way the module is used as a component.

The *model* is the kernel's basic structuring unit. It is an abstraction of some dynamic behaviour. A model consists of three parts: terminals, parameters and realizations. The terminals are variables which constitute a well-defined interface to describe interaction with the environment. Parameters are interface variables defined by the model designer to allow the user to adapt the description of behaviour.

Realizations

A realization is a description of model behaviour. A model user can use a model without having to bother about how its behaviour is defined internally and the model designer can and must define its behaviour without any assumptions about the environment.

One reason for treating a realization as a separate part within the model is that we want to have multiple realizations. Different realizations can give more or less refined descriptions of the behaviour or they can define the behaviour for different working conditions or phases of a batch process. The user can choose the appropriate realization for each particular use.

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. The submodels can in turn have structured realizations which means that the model 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.

Parameters

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.

To support reparameterizations and alternative parameters, it is possible to define relations between parameters. See Mattsson (1989b).

Terminals

Terminals can be viewed as variables which are shared by the internal description of the model and its environment.

It is natural to aggregate terminal variables, since the description of an interaction often involves several quantities. We propose two types of composite terminals: record and vector terminals. Their subterminals can be simple, record or vector terminals.

EXAMPLE 1—A pipe terminal
A terminal to describe the ends of a pipe or the inlets and outlets of pumps, valves and tanks can be defined as a record terminal

```
PipeTerminal IS A RecordTerminal WITH
  components:
    p IS A PressureTerminal;
    q IS A MassFlowTerminal;
    d IS A DiameterTerminal;
END;
```

having three components, which are simple terminals. The component d defines the diameter of the pipe or hole. □

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. We will not discuss user interfaces here, but just point out that a block diagram is a good way of describing model structure. Elmqvist and Mattsson (1989) have developed a prototype simulator, which exploits some features of modern computer graphics.

A connection between two structured terminals means that their first components are connected to each other and so on recursively down to the level of simple terminals. There are 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.

A simple terminal has an attribute defining the unit of measure with the SI unit as default. It is used for automatic introduction of proper scale factors in the connection equations, thus eliminating the need of user defined adapters.

It is important to note that generally the causality of a terminal (input or output) is not defined by the model designer but is inferred from the use of the model.

The semantics of a connection is kept simple, since we do not want to provide two different ways of describing complex behaviors. It is possible to describe complex interaction by introducing new submodels. It is also desirable to make the means to describe interactions independent of the frameworks used to describe the behaviour of primitive models.

EXAMPLE 2—Pipe terminals cont.
Assume that we want to model a system where a tank has a valve at the outlet. We then just connect the outlet terminal of the tank model to the inlet terminal of the valve model. The equations for the interaction saying that the pressures as well as the diameters should be equal and that the mass flows should sum to zero are deduced automatically from the connection. □

Consistency of connections

It is important to make the use of library models safe and reliable. The encapsulation of the models prevents to a large extent unintended abuse, but the terminals are dangerous holes in the wall. To allow automatic checks of connections, the model developer may add extra information, which also is useful for documentation.

Simple terminals have the attributes name of quantity 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 like English or Swedish. Information about ranges of validity is used to test for unintended abuse during simulations.

A terminal component may be declared as time-invariant. Such a terminal is similar to a parameter. This has two complementary uses. First, a connection implies automatic propagation of parameter values from one submodel to another. Second, if the two connected parameter terminals have defined values, they must be equal for the connection to be consistent.

EXAMPLE 3—Pipe terminals cont.
Consider PipeTerminal in Example 1. The pressure component p can be defined by

```
PressureTerminal IS A SimpleTerminal
WITH
  attributes:
    value       := UNKNOWN;
    quantity    := pressure;
    unit        := kPa;
    direction   := across;
    variability := time_varying;
    causality   := UNKNOWN;
END;
```

The mass flow component `q` and the diameter component `d` are defined in analogous ways. An important difference is that mass flow is a through variable and the direction attribute should be set to `in` or `out`.

The variability of `d` ought to be set to `time_invariant` if the model does not allow the size of the pipe or hole to vary with time. It also allows automatic check of that two connected pipes are of the same diameter.

The terminal could also have a component indicating medium, which can be used for consistency checking or parameter propagation. For example, we can check that water pipes are connected to water pipes. □

Unspecified terminal attributes

To allow exploratory model development and prototyping, a declaration of a terminal may leave attributes unspecified as long as necessary information can be deduced from the context. Unspecified attributes make it possible to develop generic models. To support consistency checks of generic models, the model developer can specify relations between unspecified attributes. See Mattsson (1989b).

Object-oriented representation

In this section we will outline the conceptual design of a kernel for model representation. The basic entities, relations between entities and operations on them are discussed.

Object-oriented programming has been an increasingly popular methodology for software development. Increased programmer productivity, increased software quality and easier program maintenance are the objectives for this new methodology. Object-oriented programming supports these objectives by facilitating modularization and reuse of code. We will here show that ideas from object-oriented programming are useful also for model representation. For a brief introduction to object-oriented programming see Stefik and Bobrow (1986).

Basic model objects

Models and model components are *objects* in the kernel for model representation. An object has a unique identity within the system and it contains a collection of attributes. There is a number of important types of objects recognized in the kernel. They are representations of model structuring entities discussed in the previous section:

- models,
- terminals,
- parameters and
- realizations.

The last three object types can be used as components of models.

Class objects and relations

In our proposal, all model objects are represented as classes. In object-oriented programming a class describes the properties common to a set of similar objects – it defines an object *type*. For this reason, a model defines a component type rather than a particular instance of a component; the same applies to realizations, terminals, etc. A class can have a number of *attributes* which can be simple variables or relations to other model objects.

There are a three important relations which can be established between model objects. These are:

- has – part-of
- subclass – super class
- connection

The *has*-link is typically used between a model and its terminals, parameters and realizations. Further, a structured realization has this kind of relation to other models indicating the submodels. A has-link is stored as an attribute of the owner. The inverse relation is called *part-of*.

One class can be defined to be a *subclass* of another class – the super class. The subclass will inherit all properties of the super class in addition to the locally defined properties. *Inheritance* is an important concept in object-oriented programming and its use in this context will be discussed below.

A *connection* is a symmetric relation between two terminals and it is stored as an attribute of a structured realization.

EXAMPLE 4—Tank model
In this example we will show a model of a tank written in Omola (Object-Oriented Modelling Language) (Andersson, 1989a,b). Omola is a declarative language for model representation that has been designed to support our proposed concepts.

```
Tank IS A Model WITH
 terminals:
  inlet  IS A InPipeTerminal;
  outlet IS A OutPipeTerminal;
  level  IS A OutTerminal;
 parameters:
  area TYPE real := 1.0;
  roh  TYPE real := 1.0;
 realization:
```

```
  normalBehaviour IS A SetOfDAE WITH
   equations:
    area*dot(level) =
      inlet.q - outlet.q;
    inlet.p + level*roh*g =
      outlet.p - roh*v*abs(v)/2;
    outlet.q =
      pi*(outlet.d/2)^2*v*roh;
 END;
END;
```

This code represents a tank model with three terminals, two parameters and a realization component stored as attributes. The inlet and outlet terminals are both pipe terminals as in Example 1, but with directed flow components. For `inlet` positive flow is into the tank and for `outlet` positive flow is out from the tank. The realization has three equation attributes. The first equation is a mass balance and the other two are derived from Bernoulli's equation. In Figure 1 we can see some of the objects involved and their relations represented graphically. □

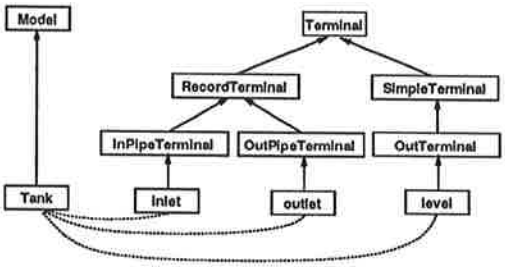

Figure 1. Some of the objects and their relations in the tank model. Subclass links are solid while has-links are dashed.

Inheritance

Inheritance is an intricate but powerful concept in object-oriented programming. When a class is defined to be a subclass of another class it will inherit all attributes and properties from the super class. The subclass is then free to add additional attributes or to redefine inherited attributes. Inheritance can be used to separate out some general attributes from a set of similar classes into a common super class.

Inheritance will facilitate reuse of models since carefully designed general models can be saved in libraries. These models or model components can be used as super classes of more specialized model objects. We have already seen how terminals have been defined in this way. The `inlet` and `outlet` terminals of the tank model are subclasses of `InPipeTerminal` and `OutPipeTerminal` which are specializations of the same super class `RecordTerminal`.

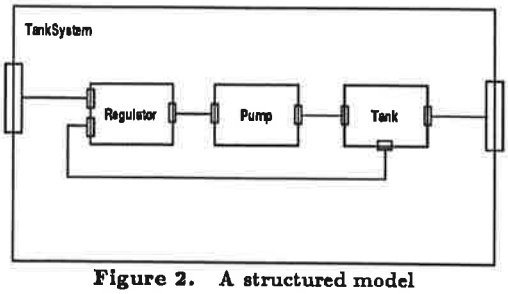

Figure 2. A structured model

As an example of how models can be defined by specializations we can imagine a model of a regulator defining only the terminals: set-point, measure value and control value. This model can be specialized into different types of regulators by means of adding different realizations. We may then define a structured model like in Figure 2, containing the most general regulator model. The structured model can then be specialized to contain different regulator models.

Interpretation of model objects

Model structures represented in the kernel or in Omola code can be accessed and manipulated by different tools in a CACE environment. We may say that a particular tool that extracts relevant properties of a model *interprets* the model. Different tools may extract different properties and therefore, they interpret the model differently.

Since all model objects discussed so far are classes, i.e., they represent types rather than instances of model objects, one obvious interpretation is to use a model as a template to create a *model instance*. A model instance is, for example, needed when the model is going to be simulated. Then there must be representations for each particular model object and state variable. The instantiation procedure is recursive in the components and submodels. Typically when we want to simulate a model it is first instantiated then all equations are extracted from the primitive models and equations are generated from the connections. Second, the equations are sorted and turned into code that can be used by the DAE-solver. Since the model structure is maintained in the simulation model (the model instance) the user can access it the normal way, perhaps through its block diagram, and examine or change parameters and initial values.

As examples of other possible interpretations of model objects we can mention

- to generate a graphical picture of a system structure,
- to generate a text descriptions of a model for documentation,
- to generate a special purpose code, e.g., regulator code or

- to turn a model into a form accepted by a particular design package.

Conclusions

In this paper we have outlined some basic concepts supporting model development and reuse. The issues we have been focusing on are:

- A kernel serving as a central model data base in an integrated environment for model development, simulation, analysis, design, documentation etc.
- Declarative and equation based behaviour descriptions to make the models versatile and useful for various applications.
- Hierarchical models with well defined interfaces based on terminals and parameters.
- Terminal attributes for automatic check of connection consistency making the use of library models safer.
- An internal representation which preserves the structure of models.
- Multiple realizations supporting model versions and alternative behaviour.
- Models represented as classes with inheritance facilitates reuse and incremental model development.

Much could be gained if we could agree upon a common set of ideas. It is time to lay the foundation for a new standard for model representation. IFAC has a working group on standards for CACSD Software. This group has not addressed non-linear systems yet, but it has focused on linear systems. It may be remarked that to build flexible model libraries we must also agree on common principles for model development. This is a hard task, but it might be possible to achieve in certain application areas.

The kernel is a result of a project to develop concepts and tools for model development and simulation in CACE. A prototype implementation has been written in Common Lisp and KEE[1]. 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. To make are results more generally available, an implementation in C++ is under development.

Acknowledgements

The authors would like to thank Professor Karl Johan Åström, 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.

[1] Knowledge Engineering Environment, KEE is a trademark of IntelliCorp, Inc.

References

ANDERSSON, M. (1989a): "An Object-Oriented Modeling Environment," *Proc. 1989 European Simulation Multiconference, Rome, June 7–9, 1989*, The Society for Computer Simulation International, pp. 77–82.

ANDERSSON, M. (1989b): "Omola – An Object-Oriented Modelling Language," Report TFRT-7417, 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.

KHEIR, N.A. (Ed.) (1988): *Systems Modeling and Computer Simulation*, Marcel Dekker, Inc., New York.

KREUTZER, W. (1986): *System Simulation – Programming styles and languages*, International Computer Science Series, Addison-Wesley.

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. 1989 European Simulation Multiconference, Rome, June 7–9, 1989*, The Society for Computer Simulation International, pp. 63–68.

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.