Proceedings of the 32nd Conference
on Decision and Control
San Antonio, Texas • December 1993

MODELING AND SIMULATION OF BEHAVIOURAL SYSTEMS

Sven Erik Mattsson, Mats Andersson and Karl Johan Åström

Department of Automatic Control, Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
svenerik@control.lth.se

Abstract

This paper describes a modeling methodology that is matched to behavioural system descriptions. The approach covers a wide range of systems occurring in automatic control. A modeling language called Omola and an environment of computer tools called OmSim support the methodology. Models can be decomposed hierarchically with well-defined interfaces for describing interactions. All components are represented as classes. Inheritance and specialization support easy modification. Behavioral descriptions are given in terms of differential-algebraic equations, difference equations and elements of discrete events.

1. Introduction

Although the state space paradigm has been very successful there is also a need for other approaches. The behavioural approach of Willems [16, 17] has several advantages in physical modeling. Particularly it avoids causality issues which is significant when developing model libraries. This is important because of a significant industrial interest in mathematical models and simulation. The decrease in the cost of computation makes simulation more useful as a general-purpose tool. The problem is that modeling is difficult and time consuming today. Most new users who are interested in using simulation to solve their problems find today's simulation tools to be inadequate and difficult to use. There are very few general tools for modeling. Specialized modeling tools have been developed in different branches of engineering. However, it is necessary for a control engineer to have a unified approach to modeling, since a control system is often composed of parts covering a wide variety of technologies. As an example consider a robot. It is a mechatronic system. There are good special-purpose tools for rigid body systems and electrical systems. However, it is extremely difficult to use them to produce a model of the total robot. There is no standard for model representation. Furthermore, many special-purpose tools cannot produce explicit models.

Today's most used continuous-time simulation languages such as ACSL, CSMP, EASY5, Simnon and SIMULINK are not well-adapted to model representation. These languages are not modeling languages, but languages to describe simulation problems. Their mathematical frameworks for problem specification follow the CSSL standard from 1967 [15]. It was then necessary to choose the problem representation to allow efficient numerical solution. The task of converting the user's model and input data to this problem representation, which often requires a significant effort in terms of analysis and analytical transformation, was left to the user. Since simulation is used to solve different problems in feasibility studies, for design and for decision support, the users have to transform the model differently for each type of problem.

Computer and software technology has now progressed so much that it is possible to make a drastic departure from earlier approaches to modeling and simulation. Numerical methods that can solve more general problem have also been developed. There is a great potential of using symbolic analysis and manipulation to automatically convert high-level representations to representations that are more efficient for numerical solution.

Omola and OmSim

As a modest step to support modeling we have developed a new general modeling language called Omola [1]. Omola supports object-oriented structuring concepts. Models can be decomposed hierarchically with well-defined interfaces for describing interaction. All model components are represented as classes. Inheritance and specialization support easy modification. Omola supports behavioural descriptions in terms of differential-algebraic equations (DAE) and difference equations. The behavioural descriptions may also contain discrete event elements. The primitives for describing discrete events allow implementation of high level descriptions as Petri nets and Grafcet. OmSim is an environment of tools supporting modeling and simulation of Omola models.

The paper is organized as follows. In Sections 2 and 3 we discuss mathematical frameworks to describe behaviour and methods for simulating DAE systems. The model structuring concepts of Omola are introduced by means of an example in Section 4.

2. Mathematical Frameworks

Ordinary differential equations on explicit state space form $\dot{x} = f(x, u)$ and $y = g(x, u)$, where u is input and y is output, has traditionally been the underlying mathematical framework in continuous-time simulation. However, it is an emerging consensus that differential-algebraic equation (DAE) systems is the natural framework. An early modeling language that allowed non-causal submodels and differential equations on a general implicit form was Dymola [6]. In the area of chemical process simulation there are early tools, e.g. SPEEDUP, supporting use of differential-algebraic equations. The use of genuine equations has a long tradition in static modeling of chemical processes. Non-causal models is also one of the premises of the behavioral approach to dynamical systems proposed by Willems [16, 17].

In process modeling, the physical components are normally represented by DAEs while the supervisory systems and the digital control systems are conveniently modeled by difference equations and discrete events. Unfortunately, there is no uniform theory for discrete event dynamical systems but various approaches are used to describe behavior. Grafcet, which is adapted as a standard by the International Electrotechnical Commission (Publication 848, 1987) for representing logic controllers graphically, is of basic interest for control engineers.

3. Simulation of DAE systems

It is more difficult to solve DAEs than ODEs, but the basic techniques are available. Unfortunately, today's numerical DAE-solvers are not able to solve all mathematically well-defined problems. For example, they fail to solve high-index problems. Symbolic techniques can be used to remove difficulties and to transform the problems into a form more suitable for numerical solution. Symbolic techniques are also useful to detect model errors at an early stage, where it is easier to give good error messages.

When dealing with large problems, symbolic techniques focussing on the structure i.e., which variables appear in each equation rather than how they appear are of special interest. These techniques originate from graph-theory and sparse matrix techniques. For an introduction see [5, 8]. Permutations and partitions of different kinds are typical operations. There are efficient algorithms which can be used on large problems to find out if a problem is structurally singular and to order variables and equations into a sequence of irreducible subproblems. This partition is useful to find time invariant parts and to sort out algebraic parts which represent calculation of outputs that are not needed when solving the dynamical part [10]. Descriptions of interactions between components often result in simple equations of the type $a = const \cdot b$, which are easy to use to eliminate variables. It is easy to evaluate constant subexpressions and to find and to exploit common subexpressions to decrease the computational load at simulation.

In many applications it is a feasible approach to try to transform the problem analytically into explicit state space form $\dot{x} = f(t, x)$ and to use a numerical ODE solver [4]. However, it is not a general method, since it may require analytical solutions to non-linear equations. Even if the equations are linear in the derivatives and the unknown algebraic variables it is not always feasible to transform the problem into explicit state space form, since the inverse of a sparse matrix may be dense. Models of power grids have this property. The transmission lines introduce many linear algebraic equations for voltages and currents.

Numerical DAE solvers

Numerical solution of DAEs is discussed in [2, 7]. There are numerical DAE-solvers which treat problems of the type $g(t, x, \dot{x}) = 0$, if they are provided with a routine that calculates the residual $\Delta = g(t, x, z)$ when the arguments t, x, and z are known. This is quite analogous to the use of ODE solvers for $\dot{x} = f(t, x)$, where a routine for calculating $f(t, x)$ should be provided.

A good numerical DAE solver is DASSL developed by Petzold [2]. The code for DASSL is available in public domain. It has a reputation of being one of the best and most robust solvers for DAEs. DASSL converts a DAE into a non-linear algebraic equation system by approximating derivatives with backward differences. For example, the simplest discretization is backward Euler, $\dot{x}_{n+1} \approx (x_{n+1} - x_n)/h$, where h is the step size. It turns the DAE problem into the implicit recursion $g(t_n, x_n, (x_n - x_{n-1})/h) = 0$. DASSL uses backward differences of order up to five. The version DASRT can detect zero crossings of user defined indicator functions. This is useful for continuous-time simulation with discrete events.

One drawback with multi-step methods is that discontinuities force the methods to restart with low order approximations. If one-step discretizations of Runge-Kutta type is used, this problem is avoided. Since many real problems are discontinuous or involve sampled parts or other discrete event elements, there is an interest in using implicit Runge-Kutta methods. One available high-quality numerical solver is RADAU5 [7].

High Order Index Problem

When solving an initial value problem $\dot{x} = f(t, x)$ the task is to integrate, but when solving a DAE problem, we may also need to differentiate.

Available numerical DAE solvers are designed to integrate, i.e., calculate x from $\dot{x}$, but if the problem involves differentiation, i.e., calculate $\dot{x}_i$ from x_i for some components, they are in trouble. In numerical integration it is assumed that we can make the errors arbitrarily small by taking sufficiently small steps. An integration routine typically estimates the errors by taking steps of different sizes and comparing the results. In numerical differentiation it is well-known that the step cannot be taken infinitely small. There is a finite, optimal value. Thus the error estimates calculated by the DAE solvers behave irregularly and today's DAE solvers cannot handle the situation, but they have to exit with some error message.

It is often necessary to differentiate when solving inverse dynamics problems. However, a modular approach to modeling also often results in problem formulations involving differentiations. For example, when building model libraries for mechanical components it is natural to model the components as completely free objects in a three dimensional space. When these components are used to model a robot, their motion is constrained by the mechanical couplings which require that the parts move together. The mechanical couplings produces reaction forces and reaction torques, which can be calculated by differentiating the positional constraints.

The need to differentiate is usually classified by the DAE index. The minimum number of times that all or part of $g(t, x, \dot{x}) = 0$ must be differentiated with respect to t in order to determine $\dot{x}$ as a continuous function of x and t defines the ***index*** [2, p. 17]. For example, an ODE on state space form, $\dot{x} = f(t, x)$ is index 0, and the problem defined by $\dot{x} = f(t, x, y)$ and $g(t, x, y) = 0$ is index 1 if the Jacobian $\partial g/\partial y$ is nonsingular. The available numerical DAE solvers may solve some index 2 problems, but they fail when index is greater.

High index DAE problems are natural when we try to support modeling. The index is not an problem invariant, but can be decreased by manipulations. There is an efficient algorithm based on graph-theory to find out how many times each equation should be differentiated [14]. The new differentiated problem obtained in this way is called the underlaying ODE, or UODE. However, it is often less satisfactory to solve the UODE, because its set of solutions is larger. The algebraic relations of the original DAE problem is then only implicit in the UODE as solution invariants. Unless linear, these invariants are generally not preserved under discretization. As a result, the numerical solution drifts off the algebraic constraints; the parts of the robot move apart during the simulation. Methods to obtain a low-index formulation, with a solution set identical to that of the original problem is discussed in [11], where a new general combined symbolic and numeric method is proposed.

4. Structuring Concepts

To understand large systems 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 and development of libraries of reusable components.

To illustrate and to discuss the structuring concepts of Omola, we will consider the problem in [3, pp. 75–77], where the task is to study the effects of connecting a power generator to a power line.

The model library

The exercise system can be broken up into a set of connected standard electrical components. Models of them are typically available as a result from previous modeling activities within the same application domain.

When a library for a new domain of application is to be developed, it is good to start by defining a set of *terminal* classes. A terminal is an object used for the interaction between submodels. It represents model variables which are accessible to the environment of the model and can be connected to other submodels. A terminal may represent a single physical quantity, e.g., a voltage measurement, or a set of quantities, e.g., a current and a voltage at a certain point in an electrical circuit. A common set of terminal definitions, used in all components in the library, facilitates compatibility of submodels.

The standard terminal class in the electrical library is the `ElectricTerminal` defined as:

```
ElectricTerminal ISA RecordTerminal WITH
  V ISA VoltageTerminal;
  I ISA CurrentTerminal;
END;
```

Omola is based on class definitions. Classes are used for representing models, terminals, variables and parameters. A class can have attributes which are other classes, simple variables, equations and connections. All classes are organized in a tree structure such that every class, except one predefined class, has a superclass. A class inherits the attributes of its superclass. The definition for `ElectricTerminal` says that `ElectricTerminal` has the superclass `RecordTerminal`, which is a predefined class in Omola for defining structured terminals. It says also that `ElectricTerminal` has two attributes or components `V` and `I`.

The class `VoltageTerminal` is defined as:

```
VoltageTerminal ISA SimpleTerminal WITH
  unit := "V";
  quantity := "electric.potential";
END;
```

where `SimpleTerminal` is a predefined class:

```
SimpleTerminal ISA Terminal WITH
  value TYPE Real;
  quantity TYPE String := "number";
  unit TYPE String := "1";

  variability TYPE (TimeVarying, Parameter)
    := 'TimeVarying;
  default TYPE Real;
  causality TYPE (Undefined, Input, Output)
    := 'Undefined;
END;
```

The attribute `value` holds the value. The attributes `quantity` and `unit` specify the physical quantity and unit of measure represented by the terminals. They are used by the OmSim environment to check consistency of terminal connections and to introduce proper scale factors in the connection equation. The naming of quantities is based on the international standard ISO 31. A connection between two simple terminals implies that they should be equal. A connection implies no explicit causality. However, causality can be imposed by specifying `causality` to be `Input` or `Output`, which is useful for documentation and to get automatic consistency checking when for example, a terminal represents a pure measurement. The definition of `VoltageTerminal` just redefines the values of the `quantity` and `unit` attributes.

The class `CurrentTerminal` is defined as

```
CurrentTerminal ISA ZeroSumTerminal WITH
  unit := "A";
  quantity := "electric.current";
  direction := 'In;
  default := 0;
END;
```

The part of OmSim that generates equations from terminal connections recognizes terminals that are subclasses of `ZeroSumTerminal` and generates a zero sum equation for each connected group of current terminals. The attribute `direction` specifies the positive flow direction. The current terminal has an attribute, `default`, with a zero value, which is used if the terminal is not connected.

Let us now consider the component models. Models for resistors, capacitors, inductors, voltage generators and switches are needed. All these components have in common that they have two electrical terminals and that the currents in these terminals are equal but of opposite direction. This is exploited in the library such that these components are all derived from a common super class called `TwoPole` defined as:

```
TwoPole ISA ElectricModel WITH
  T1 ISA ElectricTerminal;
  T2 ISA ElectricTerminal;
  T1.I + T2.I = 0;
END;
```

The resistor, capacitor and inductor models are simple specializations of `TwoPole` adding equations for the voltage drop. For example, the capacitor is defined as

```
Capacitor ISA TwoPole WITH
  C ISA Parameter WITH default:=1.0; END;
  I = C*(T1.V - T2.V)';
END;
```

The single quote operator in the equation gives the time derivative of the voltage difference.

Finally we will take a look at the switch model:

```
DelayedSwitch ISA TwoPole WITH
  Close, Init ISAN Event;
  Delay ISA Parameter WITH
    default:=0.0; END;
  is_closed TYPE DISCRETE Integer;
  0 = IF is_closed THEN T1.V-T2.V
      ELSE T1.I;
  OnEvent Init DO
    is_closed := 0;
    schedule(Close, Delay);
  END;
  OnEvent Close DO
    new(is_closed) := 1;
  END;
END;
```

The model represents a switch that is first open and then closes after a certain time delay. The delay time is set by a parameter. This simple behaviour is enough for this exercise. The switch model has an internal state determining if the switch is currently open or closed. A conditional equation defines either the current to zero or the voltage drop to zero. The switch is controlled by discrete events. One event, called `Init`, occurs when the model is initialized at the beginning of a simulation. It sets the switch to open and schedules another event, called `Close`, to occur after a time delay to close the switch.

The library of electrical components can be viewed as an inheritance tree by a special graphical tool in OmSim. This is shown in Figure 1.

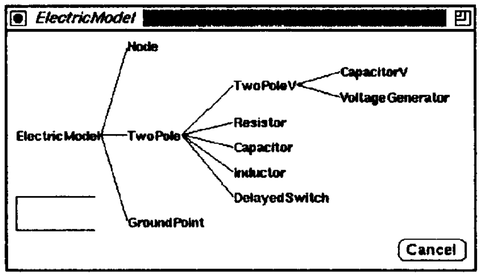

Figure 1. Inheritance tree for an electrical component library

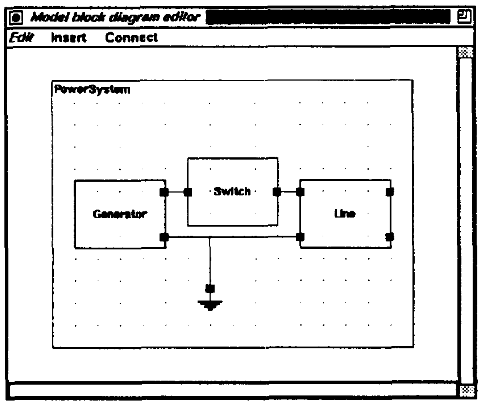

Figure 2. Decomposition of the total system

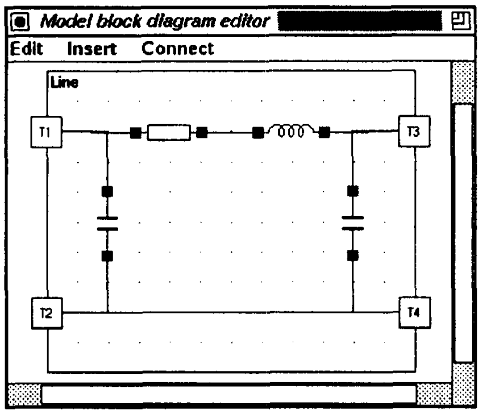

Figure 3. The power line model

Creating the structured model

When a suitable library of terminals and primitive components has been built, the actual composite model can be defined. A complex model is typically hierarchically decomposed at several levels.

The power system we want to simulate can be decomposed into three subsystems: the generator, the switching system and the power line. The structure can be seen in Figure 2. The generator is simply modeled as an ideal sine generator behind an impedance. The power line is decomposed into four primitive components in a π configuration as shown in Figure 3. The switch system is just a simple switch of the type defined above.

The subsystems and the complete model is defined in OmSim using a graphical object diagram editor. Interaction between submodels is defined by drawing connection lines between the terminals of the submodels. Figure 3 shows the model editor when the power line model has been defined. When a new model is defined using the graphical editor in OmSim, the standard graphical representation when this

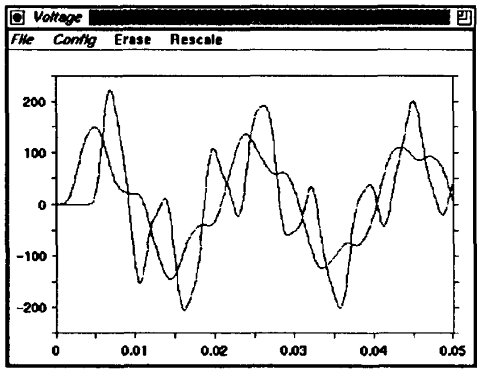

Figure 4. Result of simulation with the switch closed at 0 and 4 ms

model is used as a component in some other model, is an annotated box with small filled squares representing the terminals. An example is seen in Figure 2. It is also possible to define bitmap icons using standard bitmap editors and use them as a representation of a class of submodels. This is has been done for the library components of this example which uses standard electric diagram symbols as in Figure 3.

Simulating the system

The model is now ready to be simulated. The simulator tool in OmSim checks the model for consistency, extracts variables and equations and manipulates and transforms the problem into simulation code. A simulation result is shown in Figure 4.

Modeling of a refined system

We are now interested to study the behaviour of the system if we replace the simple switch with a time varying shunt resistor. We build a model called `ShuntSwitch` for this new switch system. Instead of redefining the complete system using the new switching submodel, we can instead make use of inheritance. A new model called `ShuntSwitchPowerSystem` is defined using `PowerSystem` as a super class. The new model inherits all subsystems and connections from the super class, except for the switching subsystem which is overwritten by the new shunt resistor switch:

```
ShuntSwitchPowerSystem ISA PowerSystem
WITH
  Switch ISA ShuntSwitch;
END;
```

There are advantages of using inheritance when defining different versions of a model. One advantage concerns model maintenance. If a subsystem, e.g., the generator, in the basic `PowerSystem` is replaced by some other generator model. This change will automatically affect all versions derived from `PowerSystem`. It is also obvious from the definition that `ShuntSwitchPowerSystem` is just a simple modification of `PowerSystem`.

5. Conclusions

In control engineering it is necessary to deal with a variety of systems in a unified manner. In this paper we have presented an approach to behavioural modeling based on a new general modeling language, Omola and a new integrated environment for model development and simulation, OmSim. It is a significant improvement compared to the approaches used in current simulators for control systems.

When supporting model development, reuse is a key issue. It should be possible to use a model to solve different problems and it should be easy to modify existing model components to describe similar systems. To implement this Omola provides object-oriented structuring concepts and equations for describing behaviour. Any approach to support use of continuous time library models demands differential-algebraic equations to describe behaviour. The use of differential-algebraic equations to describe behaviour requires in most cases symbolic manipulation to transfer the problem to a form more suitable for numeric solution. For example, it is necessary to reduce the DAE index. OmSim provides such facilities. To support safe and reliable use of library models, Omola has language constructs allowing automatic consistency checking.

OmSim has been used in a couple of application projects in the areas of power systems and chemical processes [9, 12, 13]. It is a prototype environment and not a full-fledged professional and commercial product. The aim of the project has been to implement an environment which can be used in academia and industry for feasibility studies and as a basis for further research and commercial products. OmSim currently runs on Sun-4 workstations under the X Window System. It is implemented in C++ and uses only public domain software. OmSim is available via anonymous FTP.

Acknowledgements

The development of Omola and OmSim has been supported by the Swedish National Board for Industrial and Technical Development. Large parts of OmSim have been designed and implemented by Dag Brück and Tomas Schönthal.

6. References

[1] M. ANDERSSON. *Omola—An Object-Oriented Language for Model Representation.* Lic Tech thesis TFRT-3208, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, May 1990.

[2] K. E. BRENAN, S. L. CAMPBELL, and L. R. PETZOLD. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations.* North-Holland, Amsterdam, 1989.

[3] F. CELLIER. *Continuous System Modeling.* Springer-Verlag, New York, 1991.

[4] F. CELLIER and H. ELMQVIST. "Automated formula manipulation supports object-oriented continuous-system modeling." *IEEE Control Systems,* **13:2**, pp. 28–38, April 1993.

[5] I. S. DUFF, A. M. ERISMAN, and J. K. REID. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[6] H. ELMQVIST. *A Structured Model Language for Large Continuous Systems.* PhD thesis TFRT-1015, Depatment of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[7] E. HAIRER, C. LUBICH, and M. ROCHE. *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods.* Lecture Notes in Mathematics No. 1409. Springer-Verlag, Berlin, 1989.

[8] R. S. H. MAH. *Chemical Process Structures and Information Flows.* Butterworths, Boston, 1990.

[9] S. E. MATTSSON. "Modelling of power systems in Omola for transient stability studies." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design, CADCS '92,* Napa, California, March 1992.

[10] S. E. MATTSSON, M. ANDERSSON, and K. J. ÅSTRÖM. "Object-oriented modeling and simulation." In LINKENS, Ed., *CAD for Control Systems,* pp. 31–69. Marcel Dekker, Inc., 1993.

[11] S. E. MATTSSON and G. SÖDERLIND. "Index reduction in differential-algebraic equations using dummy derivatives." *SIAM Journal of Scientific and Statistical Computing,* **14:3**, pp. 677–692, May 1993.

[12] B. NILSSON. "Object-oriented chemical process modelling in Omola." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design, CADCS '92,* Napa, California, March 1992.

[13] B. NILSSON. *Object-Oriented Modelling of Chemical Processes.* PhD thesis ISRN LUTFD2/TFRT--1041--SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, September 1993.

[14] C. C. PANTELIDES. "The consistent initialization of differential-algebraic systems." *SIAM Journal of Scientific and Statistical Computing,* **9:2**, pp. 213–231, March 1988.

[15] J. C. STRAUSS. "The Sci continuous system simulation language (CSSL)." *Simulation,* **9**, December, pp. 281–303, December 1967.

[16] J. C. WILLEMS. "From time series to to linear system — Part I. Finite dimensional linear time invariant systems." *Automatica,* **22:5**, pp. 561–580, 1986.

[17] J. C. WILLEMS. "Paradigms and puzzels in the theory of dynamical systems." *IEEE Transactions on Automatic Control,* **36:3**, pp. 259–294, March 1991.