1.7.2 Software for Modeling and Simulating Control Systems

Martin Otter
Institute for Robotics and System Dynamics, German Aerospace Research Establishment Oberpfaffenhofen (DLR), Postfach 1116, D-82230 Wessling, Germany, e-mail: Martin.Otter@DLR.de

François E. Cellier
Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona 85721, U.S.A., e-mail: Cellier@ece.arizona.edu

1 Introduction

Software for the simulation of continuous–time systems was first standardized in 1967 [Augustin *et al.*, 1967]. The standard committee consisted largely of control engineers. Thus, one would expect that today's simulation languages for continuous system simulation should be particularly well suited for modeling and simulating control systems. This article shall answer the question of whether this expectation holds true or not.

There has always been a strong link between the control and simulation communities. On the one hand, simulation is an extremely important tool for all and every control engineer who is doing practical control system design in industry. For arbitrarily nonlinear plants, there is often no alternative to designing controllers by means of trial and error, using computer simulation. Thus, there is hardly any control engineer who wouldn't be using simulation at least occasionally. On the other hand, although simulation can be (and has been) applied to virtually all fields of science and engineering (and some others as well), control engineers have always been among the most cherished of its customers — after all, they have paid the butter on the bread of many a simulation software designer for years. Moreover, a good number of today's simulation researchers received their graduate education in control engineering.

There exist on the market many highly successful special–purpose simulation software tools, e.g. for the simulation of electronic circuitry, or for the simulation of multibody system dynamics, and there is (or at least used to be) a good reason for that. However, there is no market to speak of for special–purpose control system simulators, in spite of the fact that control is such an important application of simulation. The reason for this seeming discrepancy is that control systems contain not only a controller, but also a plant, which can be basically anything. Thus, a simulation tool that is able to simulate control systems must basically be able to simulate pretty much anything.

Hence a substantial portion of this article shall be devoted to a discussion of general–purpose simulation software. Yet, control systems do call for a number of special features in a simulation tool, and these features shall be pointed out explicitly.

This article is structured in three parts. In a first section, the special demands of control systems to a general–purpose simulation tool are outlined. In a second part, the article classifies the existing modeling and simulation tools and mention a few of them explicitly.

The article ends with a critical discussion of some of the shortcomings of the currently available simulation tools for modeling and simulating control systems.

This article is written with several different customer populations in mind. It should be useful reading for the average practical control engineer who needs to decide which simulation tool to acquire and, maybe even more importantly, what questions to ask when talking to a simulation vendor. It should, however, also be useful for simulation software vendors who wish to upgrade their tools to better satisfy the needs of an important subset of their customer base, namely the control engineers. It should finally appeal to the simulation research community by presenting a state–of–the–art picture of what has been accomplished in control system simulation so far, and where some of the still unresolved issues are that might be meaningful to address in the future.

2 Special Demands of Control Engineers to a Simulation Tool

This section shall discuss special requirements of control engineers as far as simulation tools are concerned.

2.1 Block Diagram Editors

Block diagrams are the most prevalent modeling tool of the control engineer. Figure 1 shows a typical block diagram of a control loop around a single–input single–output (SISO) plant. Evidently, control engineers would like to describe their systems in the simulation model in

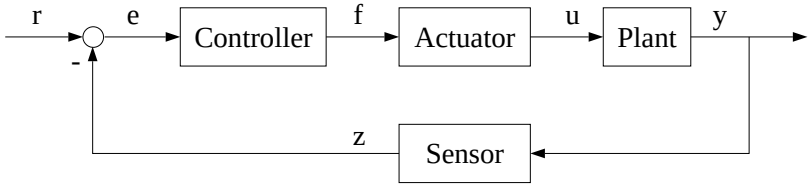

Figure 1: Typical control loop around a SISO plant.

exactly the same fashion. After all, a "model" is simply an encoded form of the knowledge available about the system under study.

Why are block diagrams so essential to control engineers? Most control systems of interest are man–made. Thus, the control engineer has a say on how the signals inside the control system influence each other. In particular, control engineers have learned to design their control systems such that the behavior of each block is, for all practical purposes, independent of the block(s) that it feeds. This can be accomplished by placing voltage follower circuits in between neighboring blocks, as shown in Fig. 2.

In the block diagram, these voltage follower circuits are never actually shown. They are simply assumed to be present at all times. Control engineers do this because it simplifies the control system analysis, and thereby indirectly also the control system design. Furthermore,

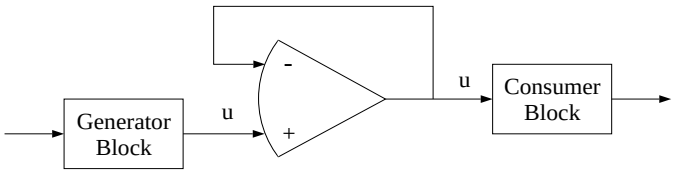

Figure 2: Voltage follower circuit for decoupling neighboring blocks.

it is exactly the same mechanism that helps with decoupling the reaction of the control system to different control inputs. If you depress the gas pedal of your car, you want your car to speed up, and not make a left turn at the same time.

Evidently, it is possible to use block diagrams also to describe any other type of system, such as the simple electrical circuit shown in Fig. 3. However, this is an *abuse* of the concept

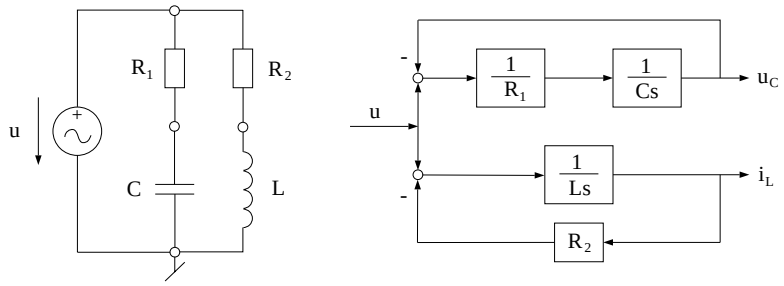

Figure 3: Electrical circuit described as block diagram.

of a block diagram. Signals, such as the electrical potential on and the current through a wire, that physically belong together and are inseparable from each other, get separated in the block diagram into two totally independent signals. Thereby, the block diagram loses all of its topological significance.

The article entitled "Block Diagrams" by Frederick and Close in this handbook discusses explicitly the use of block diagrams in control engineering. Because control engineers are so important to them, most simulation software vendors offer today a block–diagram editor as *the* graphical modeling tool. Unfortunately, block–diagram editors are not a panacea for all graphical modeling needs. Block–diagram editors are certainly not the right tool to describe e.g. electrical circuits or multibody systems.

2.2 Hierarchical Modeling

Control systems are frequently built like an onion. One control loop encompasses another. For example, it is quite common that a local nonlinear control loop is built around a nonlinear robot arm with the purpose of *linearizing* the behavior of the robot arm, such that, from the outside, the arm with its local controller looks like a linear system. This system then becomes a block in another control system at a hierarchically higher level. The purpose of that control layer may be to *decouple* the control inputs from each other, such that each control input drives precisely one link (the physical configuration may be different). This control system then turns into a series of blocks in a yet higher–level control configuration, in which each individual control input is controlled for performance.

Evidently, control engineers would like their block–diagram editors to behave in exactly the same fashion. One entire block diagram may become a single block at the next higher level. Most of the block–diagram editors currently on the market offer such a hierarchical decomposition facility.

2.3 Plant Modeling

One part of the control system to be simulated is the plant to be controlled. As was mentioned earlier, this plant can represent anything that is controllable. It can be a thermal power plant, or an aircraft, or an aquarium for tropical fish. In order to test his or her controller design, the control engineer should be able to simulate the plant with the control system around it.

As was mentioned before, block diagrams are hardly the right tool to model a physical plant. In the article entitled "Modeling from Physical Principles" by Cellier *et al.* in this handbook, the problem of modeling physical systems is discussed in greater detail.

However, it should still be mentioned that also the controllers, after having been designed in an abstract fashion, need to be implemented using physical components. Although the control engineer can choose these components, they may still have some non–ideal characteristics, the effect of which ought to be analyzed before the system is actually built. In this case, even the controller blocks become physical systems, and the same restrictions that were previously mentioned with respect to the physical plant to be controlled apply to them as well.

In summary, block diagrams are only useful to describe the higher levels of the control system architecture, but are rarely a good choice for describing the physical layer at the bottom of the hierarchy.

2.4 Coupling Models From Different Sources

Control systems are often interdisciplinary. A car manufacturer may want his control engineers to simulate the behavior of the engine before the new model is ever built. However, the engine contains the fuel delivery system, the electrical spark plugs, fans that blow air after being driven by some belts that are hooked to the mechanical subsystem, the thermal comportment of the engine, and vibrations produced by the interaction between the various components, to mention just a few of its facets.

Simulation is, in practice, mostly done to save money. If it takes more time to build a simulation model than to build the real system, simulation will hardly be a viable option, because time is money.

Car manufacturers don't build their entire product from scratch anymore. If you open the hood of your American–built car, you may encounter a Japanese engine, a German transmission, and a French fuel–injection system. Car manufacturers buy many of the components of their cars from other sources. More and more often, car manufacturers request that these components be delivered together with simulation models capturing their behavior, because it cannot be expected of the control engineers working for the car manufacturer

that they first start modeling each of the second–source components separately. They would never have their simulation models ready by the time the new car model needs to be built. What cannot, however, be expected is that all these simulation models are delivered encoded in the same simulation language. The transmission may have been modeled in Adams, the electrical system in Spice, the fuel injection system in ACSL, etc.

The control engineers are at the top of the hierarchy. It is their job to ensure that all the components work properly together. Hence it is important that a control engineer can bond together models encoded in different modeling languages in a single simulation environment. This is a very tough problem.

2.5 Linearization

One way that control engineers deal with control systems is to linearize the plant to be controlled or at least a part thereof. This then enables them to perform the controller design in a simplified fashion, since there exist analytical controller design strategies for linear systems, whereas the nonlinear control system design would have to be done by trial and error.

Control engineers want the linearization of the original model to be done in an automated fashion. Moreover, this has to happen inside the modeling environment, since the original nonlinear model needs to be interpreted in this process.

This feature is very important for control engineers. They want to be able to compare the behavior of the linearized model with that of the original nonlinear model before they go about designing the controller. Then, after the controller has been synthesized, they would like to simulate the control behavior of the controller when applied to the original nonlinear plant. Finally, they may want to use the linear control system design only as a first step on the way to determining an appropriate controller for the original nonlinear plant. The so synthesized controller can be interpreted as an approximation of the ultimately used controller. It has the right structure, but the previously found parameter values are only approximations of the final parameter values. The parameters are then fine–tuned using the nonlinear plant model together with some parameter optimization algorithm.

Some of the currently available simulation environments, such as ACSL and SIMULINK, offer a limited model linearization capability. A linear model of the type:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \tag{1}$$

is obtained from the original nonlinear model:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{2}$$

by approximating the two Jacobians:

$$\mathbf{A} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \quad ; \quad \mathbf{B} = \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \tag{3}$$

through numerical differences. The facility is limited in three ways: (i) There is no control over the quality of the numerical difference approximation, and thereby over the linearization. The problem can be arbitrarily poorly conditioned. A symbolic differentiation of the

model to generate the Jacobians may be more suitable and is entirely feasible. (ii) The approximation is necessarily local, i.e., limited to an operating point $< \mathbf{x_0}, \mathbf{u_0} >$. If, during simulation, the solution starts to deviate a lot from this operating point, the approximation may be meaningless. (iii) The approximation makes the assumption that the state variables must be preserved. This assumption may be too strong. If a subsystem is represented by the state–space model:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}, \mathbf{u}, t) \end{aligned} \tag{4}$$

all one may wish to preserve is the input–output behavior, but this behavior should be preserved over an entire trajectory or even set of trajectories. This can often be accomplished by a model of the type:

$$\dot{\mathbf{z}} = \mathbf{A} \cdot \mathbf{z} + \mathbf{B} \cdot \mathbf{u} \tag{5}$$

$$\mathbf{y} = \mathbf{C} \cdot \mathbf{z} + \mathbf{D} \cdot \mathbf{u} \tag{6}$$

if only the length of the linear state vector $\mathbf{z}$ is chosen sufficiently larger than that of the original state vector $\mathbf{x}$ [Ljung, 1987].

2.6 Parameter Identification

Contrary to the plant parameters that can be determined (at least in an approximate fashion) from physical considerations, controller parameters are technological parameters that can be freely chosen by the designer. Hence a tool is needed to determine optimal controller parameter values in the sense of minimizing (or maximizing) a performance index.

Although some simulation environments offer special tools for parameter identification, they all proceed in a purely numerical fashion. The authors of this article are convinced that much can be done to improve both the convergence speed and the convergence range of the optimization by proceeding in a mixed symbolic and numerical fashion.

Let $\mathbf{p}$ be the vector of unknown parameters, and PI the performance index to be optimized. It is fairly straightforward to augment the model at compile time by a sensitivity model that computes the sensitivity vector $\partial PI/\partial \mathbf{p}$. If there are k parameters and n equations in the original model, the augmented model will have $n(k+1)$ equations.

The control engineer can then look at the magnitude of the sensitivity parameters as a function of time, and pick a subset of those (those with large magnitudes) for optimization. Let us assume the reduced set of parameters $\mathbf{pr}$ is of length $kr < k$. Optimizing $PI(\mathbf{pr})$ implies making $\partial PI/\partial \mathbf{pr} = 0$. The latter problem can be solved by Newton iteration:

$$\frac{\partial^2 PI_\ell}{\partial \mathbf{pr}_\ell^2} \cdot \boldsymbol{\delta}_\ell = -\frac{\partial PI_\ell}{\partial \mathbf{pr}_\ell} \tag{7}$$

$$\mathbf{pr}_{\ell+1} = \mathbf{pr}_\ell + \boldsymbol{\delta}_\ell \tag{8}$$

Each iteration implies solving the augmented set of the original equations and the equations partially differentiated with respect to design parameters. Even equations for the Hessian matrix (the second partial derivative) can be generated symbolically at compile time, if code is generated simultaneously that prevents these additional equations from being executed during each function evaluation.

2.7 Frequency Domain

Control engineers like to switch back and forth between the time domain and the frequency domain when they are dealing with linear (or linearized) systems.

Most simulation systems offer the capability to enter blocks as transfer functions. The polynomial coefficients are used in a set of differential equations (using the controller–canonical form), thereby converting the transfer function back into the time domain.

Although this feature is useful, it doesn't provide the control engineer with true frequency analysis capabilities. Control engineers like to be able to find the bandwidth of a plant, or determine the loop gain of a feedback loop. Such operations are much more naturally performed in the frequency domain, and it seems therefore useful to have a tool that would transform a linear (or linearized) model into the frequency domain, together with frequency domain analysis tools operating on the so transformed model.

2.8 Real–Time Applications

Control systems are often not fully automated, but represent a collaborative effort of human and automatic control. Complex systems (such as an aircraft or a power–generation system) cannot be controlled by human operators alone, because of the time–critical nature of the decisions that must be reached. Humans are not fast and not systematic enough for this purpose. Yet, safety considerations usually mandate at least some human override capability, and often, humans are in charge of the higher echelons of the control architecture, i.e., they are in control of those tasks that require more intelligence and insight, yet are less time–critical.

Simulation of such complex control systems should allow human operators to drive the simulation in just the same manner as they would drive the real system. This is useful for both system debugging as well as operator training. However, since humans cannot be time–scaled, it is then important to perform the entire simulation in real time.

Another real–time aspect of simulation is the need to download controller designs into the digital controllers that are used to control the actual plant once the design has been completed. It does not make sense to ask the control engineer to reimplement the final design manually in the actual controller, since this invariably leads to new bugs in the code. It is much better if the modeling environment offers a fully automated real–time code generation facility, generating real–time code in either C, Fortran, or Ada.

Finally, some simulators contain hardware in the loop. For example, flight simulators for pilot training are elaborate electro–mechanical devices by themselves. It is the purpose of these simulators to make the hardware components behave as closely as possible to those that would be encountered in the real system. This entails simulated scenery, simulated force feedback, possibly simulated vibrations, etc. Evidently, also these simulations need to be performed in real time.

3 Overview of Modeling and Simulation Software

There currently exist hundreds of different simulation systems on the market. They come in all shades and prices, specialized for different application areas, for different computing platforms, and embracing different modeling paradigms. Many of them are simply competitors of each other. It does not serve too much purpose to try to survey all of them. A list of current products and vendors is published in [Rodrigues, 1994].

Is such a multitude of products justified? Why are there many more simulation languages around than general–purpose programing languages? The answer is easy. The general–purpose programing market is much more competitive. Since millions of Fortran and C compilers are sold, these compilers are comparatively cheap. It is almost impossible for a newcomer to penetrate this market, because he or she would have to work under cost for too long a period to ever make it. Simulation software is sold in hundreds or thousands of copies, not millions. Thus, the software is comparatively more expensive, and individuals, who sell ten copies may already make a modest profit. Yet, the bewildering diversification on the simulation software market is certainly not overly productive.

Rather than trying to be exhaustive, the authors decided to concentrate here on a few of the more widely used products, a discussion that, in addition, shall help explain the different philosophies embraced by these software tools. This serves the purpose of consolidating the classification of modeling and simulation paradigms that had already been attempted in the previous section of this article. A more elaborate discussion of modeling and simulation software in general (not focused on the modeling and simulation of control systems) can be found in [Cellier, 1993].

3.1 Block Diagram Simulators

The natural description form of the higher echelons of a control architecture is block diagrams, i.e., a graphical representation of a system via input–output blocks (cf. also the article entitled "Block Diagrams" by Frederick and Close in this handbook). As already mentioned, most of the major simulation software producers offer a block–diagram editor as a graphical front end to their simulation engines.

Three of the most important packages of this type currently on the market are briefly discussed. All of them allow the simulation of continuous–time (differential equation) and discrete–time (difference equation) blocks and mixtures thereof. This is of particular importance to control engineers, since it allows them to model and simulate sampled–data control systems. Some of the tools also support state events, but their numerical treatment is not always appropriate. Modeling is done graphically, and block diagrams can mostly be structured in a hierarchical fashion.

- **SIMULINK** from The MathWorks Inc. [MathWorks, 1992, 1994]:
  An easy–to–use point–and–click program. SIMULINK is an extension to MATLAB, the widely–used program for interactive matrix manipulations and numerical computations in general. Of the three programs, SIMULINK offers the most intuitive user interface. MATLAB can be employed as a powerful pre– and postprocessing facility

for simulation, allowing e.g. parameter variation and optimization (although not employing the more advanced semi–symbolic processing concepts that were discussed in the previous section of this article) as well as displaying the simulation results in a rich set of different formats. SIMULINK and MATLAB are available for a broad range of computing platforms and operating systems (PC/Windows, MacIntosh, Unix/X–Windows, VAX/VMS). SIMULINK supports the same philosophy that is used within MATLAB. By default, the equations of a SIMULINK model are preprocessed into an intermediate format, which is then interpreted. This has the advantage that the program is highly interactive, and simulations can run almost at once. It has recently become possible to alternatively compile built–in elements of SIMULINK into C, to be used in the simulation or in a real–time application. However, user–defined equations programmed in the powerful MATLAB language (as M–files) are still executed many times slower due to their being interpreted rather than compiled. SIMULINK enjoys a lot of popularity, especially in academia, where its highly intuitive and easily learnable user interface is particularly appreciated.

- **SystemBuild** from Integrated Systems Inc. [Integrated Systems, 1994]:
  Overall, SystemBuild offers more powerful features than SIMULINK. For example, it offers much better event specification and handling facilities. A separate editor for defining finite–state machines is available. Models can be described by differential–algebraic equations (DAEs), and even by overdetermined DAEs. The latter are needed if e.g. general–purpose multibody system programs shall be used within a block–diagram editor for the description of complex mechanical mechanisms such as vehicles[1]. The price to be paid for this flexibility and generality is a somewhat more involved user interface that is a little more difficult to master. For several years already, SystemBuild offers the generation of real–time code in C, Fortran, and Ada. SystemBuild is an extension to Xmath (formerly MATRIX_X, the main MATLAB competitor). Xmath is very similar to MATLAB, but supports more powerful data structures and a more intimate connection to X–Windows. This comes at a price, though. Xmath and SystemBuild are not available for PC/Windows or Macintosh computers. Due to their flexibility and the more advanced features offered by these tools, these products have a lot of appeal to industrial customers, whereas academic users may be more attracted to the ease–of–use and platform–independence offered by SIMULINK.

- **EASY–5** from Boeing:
  Available since 1981, EASY–5 is one of the oldest block–diagram editors on the market. It is designed for simulations of very large systems. The tool is somewhat less easy to use than either SIMULINK or SystemBuild. It uses fully–compiled code from the beginning. After a block diagram has been built, code is generated for the model as a whole, compiled to machine code, and linked to the simulation run–time engine. This has the effect that the compilation of a block diagram into executable run–time code is rather slow, yet, the generated code executes generally faster than in the case of most other block diagram programs.

As already mentioned, block–diagram editors have the advantage that they are (usually) easy to master by even novice or occasional users, and this is the main reason for their

[1]Multibody programs that can be utilized within SystemBuild include SIMPACK [Rulka, 1990] and DADS [Smith and Haug, 1990].

great success. On the other hand, nearly all block–diagram editors on the market, including SIMULINK and SystemBuild, suffer from some severe drawbacks:

First, they don't offer a "true" component library concept in the sense used by a higher–level programming language. Especially, the user can store model components in a (so called) "library" and retrieve the component by "dragging" it from the library to the model area, with the effect that the component is being *copied*. Consequently, every change in the library requires to manually repeat the copying process, which is error prone and tedious[2].

Second, it is often the case that differential equations have to be incorporated directly in textual form, because the direct usage of block–diagram components becomes tedious. In SIMULINK and SystemBuild, the only reasonable choice is to program such parts directly in C or Fortran, i.e., by using a modeling technique from the 60s. In this respect, the general–purpose simulation languages, to be discussed in the next section of this article, offer much better support, because differential equations can be specified directly, using user–defined variable names rather than indices into an array. Furthermore, the equations can be provided in an arbitrary order, since the modeling compiler will sort them prior to generating code.

3.2 General–Purpose Simulation Languages

Block–diagram simulators became fashionable only after the recent proliferation of graphics workstations. Before that time, most general–purpose modeling and simulation was done using simulation languages that provided textual user interfaces similar to those offered by general–purpose programming languages. Due to the success of the aforementioned graphical simulation programs, most of these programs have meanwhile been enhanced by graphical front ends as well. However, the text–oriented origin of these programs often remains clearly visible through the new interface.

- **ACSL** from Mitchell & Gauthier Assoc. [Mitchell and Gauthier, 1991]:
  Available since 1975, ACSL has long been the unchallenged leader in the market of simulation languages. This situation changed in recent years due to the success of SIMULINK and SystemBuild. ACSL is a language based on the CSSL–standard [Augustin *at al.*, 1967]. An ACSL program is preprocessed to Fortran for platform independence. The resulting Fortran program is then compiled further to machine code. As a consequence, ACSL simulations always run efficiently, which is in contrast to the simulation code generated by most block–diagram simulators. User–defined Fortran, C, and Ada functions can be called from an ACSL model. ACSL can handle ODEs and DAEs, but no overdetermined DAEs. For a long time already, ACSL has supported state–event handling in a numerically reliable way (by means of the *schedule* statement), such that discontinuous elements can be handled. Recently, ACSL has been enhanced by a block–diagram front end, a post–processing package for visualization and animation, and a MATLAB–like numerical computation engine.

  A block in ACSLs block–diagram modeler can take any shape and the input/output points can be placed everywhere, contrarily e.g. to the much more restricted graphical

[2] In a higher–level programming language, a change in a library function just requires to repeat the linking process.

appearance of SIMULINK models. Consequently, with ACSL it is easier to get a closer correspondence between reality and its graphical image. Unfortunately, ACSL is not (yet) truly modular. All variables stored in a block have *global* scope. This means that one has to be careful not to use the same variable name in different blocks. Furthermore, it is not possible to define a block once, and to use several copies of this block. As a result, it is not convenient to build up user-defined block libraries. ACSL is running on a wide variety of computing platforms ranging from PCs to supercomputers. Due to the 20 years of experience, ACSL is fairly robust, contains comparatively decent integration algorithms, and many small details that may help the simulation specialist in problematic situations. Although the ACSL vendors have lost a large percentage of their academic users to SIMULINK, ACSL is still fairly popular in industry.

- **Simnon** from SSPA Systems [Elmqvist, 1975; Elmqvist *et al.*, 1990]:
  Simnon was the first direct–executing fully–digital simulation system on the market. Designed originally as a university product, Simnon is a fairly small and easily manageable software system for the simulation of continuous–time and discrete–time systems. Simnon offered, from its conception, a mixture between a statement–oriented and a block–oriented user interface. Meanwhile, a graphical front end has been added as well. Simnon has been for years a low–cost alternative to ACSL, and enjoyed wide–spread acceptance especially in academia. Due to its orientation, it suffered more than ACSL from the SIMULINK competition.

- **Desire** from G.A. & T.M. Korn [Korn, 1989]:
  Desire is another direct–executing simulation language, designed to run on small computers at impressively high speed. It contains a built–in microcompiler that generates machine code for Intel processors directly from the model specification. Since no detour is done through a high–level computer language, as is the case in most other compiled simulation languages, compilation and linking are nearly instantaneous. It is a powerful feature of the language that modeling and simulation constructs can be mixed. It is therefore easy to model and simulate systems with varying structure. For example, when simulating the ejector seat of an aircraft, several different models are simulated one after another. This is done by *chaining* several Desire models in sequence, that are compiled as needed and then run at once. Desire also offers fairly sophisticated high–speed matrix manipulation constructs, e.g. optimized for the formulation of neural network models. Desire is used both in academia and industry, and has found a strong market in real–time simulation of small– to medium–sized systems, and in digital instrumentation of measurement equipment.

3.3 Object-Oriented Modeling Languages

It had been mentioned earlier that block–diagram languages are hardly the right choice for modeling physical systems. The reason is that the block–diagram languages as well as their underlying general–purpose simulation languages are assignment statement oriented, i.e., each equation has a natural *computational causality* associated with it. It is always clear, what are the inputs of an equation, and which is the output.

Unfortunately, physics doesn't know anything about computational causality. Simulta-

neous events are always acausal. Modeling an electrical resistor, it is not evident ahead of time, whether an equation of the type:

$$u = R \cdot i \tag{9}$$

shall be needed, or one of the form:

$$i = \frac{u}{R} \tag{10}$$

It depends on the environment in which the resistor is embedded.

Consequently, the modeling tool should relax the artificial causality constraint that has been imposed on the model equations in the past. By doing so, a new class of modeling tools results. This concept has been coined the *object–oriented modeling* paradigm, since it provides the modeling language with a true one–to–one topological correspondence between the physical objects and their software counterparts inside the model. The details of this new modeling paradigm are discussed more thoroughly in the article entitled "Modeling from Physical Principles" in this handbook and shall not be repeated here.

- **Dymola** from Dynasim AB [Elmqvist, 1978, 1995]:
  The idea of general object–oriented modeling, and the first modeling language implementing this new concept, Dymola, were created by Elmqvist as part of his Ph.D. dissertation [Elmqvist, 1978]. Dymola offered already then a full topological description capability for physical systems, and demonstrated the impressive potential of this new modeling approach by means of an object–oriented model of a quite complex thermal power station. However, the demand for such general–purpose large–scale system modeling tools had not awakened yet, and neither was the computer technology of the era ready for this type of tool. Consequently, Dymola remained for many years a university prototype with fairly limited circulation. The book *Continuous System Modeling* [Cellier, 1991], which assigned a prominent role to object–oriented modeling and Dymola, reignited the interest in this tool, and since the fall of 1992, Dymola has become a commercial product. Many new features have been added to Dymola since then such as (even multiple) inheritance, a MATLAB–like matrix capability, a high–level object–oriented event–handling concept able to correctly deal with multiple simultaneous events, handling of higher–index differential algebraic equations, to mention only a few. Dymola is a model compiler that symbolically manipulates the model equations and generates a simulation program in a variety of formats including ACSL, Simnon, Desire, and SIMULINK (C–SimStruct). It also supports a simulator based on the DSblock format (to be discussed in the next subsection), called Dymosim. A graphical front end, called Dymodraw, has been developed. It is based on *object diagrams* rather than block diagrams. Models (objects) are represented by icons. Connections between icons are non–directional, representing a physical connection between physical objects. An electrical circuit diagram is a typical example of an object diagram. Also available is a simulation animator, called Dymoview, for graphical representation of motions of two– and three–dimensional mechanical bodies.

- **Omola** from Lund Institute of Technology [Andersson, 1990, 1992, 1994]:
  Omola was created at the same department that had originally produced Dymola. At the time when Omola was conceptualized, the object–oriented programming paradigm had entered a phase of wide–spread proliferation, and the researchers in Lund wanted

to create a tool that made use of a terminology that would be closer to that currently employed in object–oriented programming software. Omola is still a university prototype only. Its emphasis is primarily on language constructs, whereas Dymola's emphasis is predominantly on symbolic formula manipulation algorithms. Omola is designed for flexibility and generality, whereas Dymola is designed for high–speed compilation of large and complex industrial models into efficient simulation run–time code. Omola supports only its own simulator, called Omsim. In order to provide a user–friendly interface, Omola also offers an experimental object–diagram editor. Yet, Omola's object–diagram editor is considerably less powerful than Dymola's.

- **VHDL–A** a forthcoming IEEE standard [Vachoux and Nolan, 1993; Vachoux, 1995]: VHDL is an IEEE standard for hardware description languages. It provides a modeling language for describing digital circuitry. VHDL has been quite successful in the sense that nearly all simulators for logical devices on the market are meanwhile based on this standard. This allows an easy exchange of models between different simulators. Even more importantly, libraries for logical devices have been developed by different companies and are being sold to customers, independently of the simulator in use. The VHDL–standard is presently under revision for an analog extension, called VHDL–A [Vachoux, 1995], to include analog circuit elements. The main goal of VHDL–A is to define a product–independent language for mixed–level simulations of electrical circuits [Vachoux and Nolan, 1993]. Different levels of abstractions and different physical phenomena shall be describable within a single model. This development could be of interest to control engineers as well, since the VHDL–A definition is quite general. It includes assignment statement based input/output blocks as well as object–oriented (physical) model descriptions, and supports differential–algebraic equations. It may well be that VHDL–A becomes a standard not only for electronic circuit descriptions, but also for modeling other types of physical systems. In that case, this emerging development could gain central importance to the control community as well. However, the VHDL–A committee is currently focusing too much on language constructs without considering the implications of their decisions on efficient run–time code generation. The simulation of analog circuits (and other physical systems) is much more computation intensive than the simulation of digital circuitry. Thus, efficient run–time code is of the essence. A standard like VHDL–A would however solve many problems. First, model exchange between different simulation systems would become feasible. This is especially important for mixing domain–specific modeling systems with block–diagram simulators. Second, a new market for model component libraries would appear, because third–party vendors could develop and sell such libraries in a product–independent way. From a puristic point of view, VHDL–A is not truly object–oriented, because some features, such as inheritance, are missing. However, since VHDL–A contains the most important feature of object–oriented modeling systems, namely support for physical system modeling, it was discussed in this context.

In order to show the unique benefits of object–oriented modeling for control applications, as compared to the well–known but limited traditional modeling systems, additional issues are discussed in more detail in the following subsections.

3.3.1 Object Diagrams and Class Inheritance

The concept of object diagrams is meanwhile well understood. The former "blocks" of the block diagrams are replaced by mnemonically shaped icons. Each icon represents an object. An icon can have an arbitrary number of pins (terminals) through which the object that the icon represents exchanges information with other objects. Objects can be hierarchically structured, i.e., when the user double–clicks on an icon ("opening" the icon), a new window may pop up showing another object diagram, the external connections of which correspond to the terminals of the icon that represents the object diagram. Connections are non–directional (they represent physical connections rather than information paths), and one connection can (and frequently does) represent more than one variable.

Figure 4 shows a typical object–diagram as managed by the object–diagram editor of Dymola. Different object diagrams can use different modeling styles. The three windows

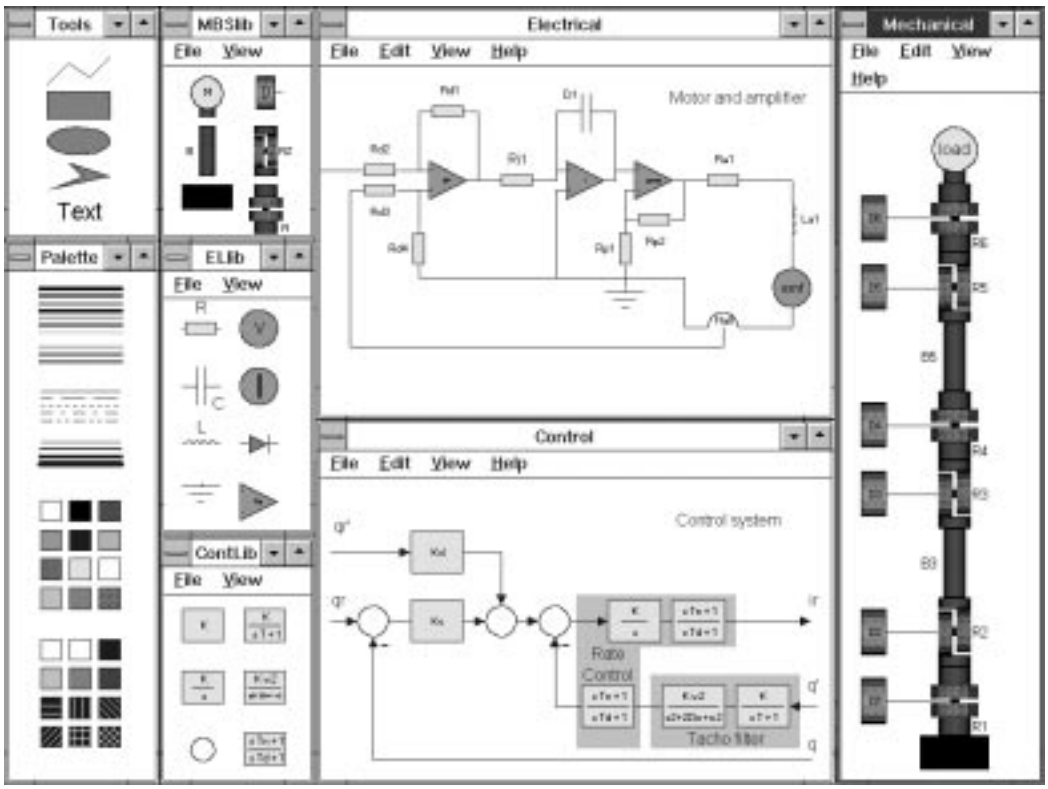

Figure 4: Object–oriented view of mechatronic model.

to the right of Fig. 4 show an electrical circuit diagram, a multibody system, and a block diagram (a special case of an object diagram). Another frequently used object–diagram representation would be a bond graph. Mechatronics systems use components from different domains, and hence it makes sense to use the modeling mechanism that is most natural to the individual domain, when modeling the different subsystems. Drive trains are attached to each joint of the robot (left part of the window *Mechanical*). A drive train class contains instances of the model classes *Control* and *Electrical*. The three windows in the second column from the left show different model libraries. Each model is represented by an icon that can be picked and dragged into the corresponding object–diagram window for composing models from components (in some cases hierarchical models by themselves) and their inter-

connections. Contrary to the case of block–diagram editors, these are true libraries in the sense that changes in a library are reflected at once (after compilation) in the models using this library. This is due to the fact that the libraries contain model classes, i.e., definitions of generic model structures, rather than the model objects themselves, and dragging an icon into an object diagram only establishes a link to the desired model class rather than leading to an object instantiation being made at once.

One important aspect of object–oriented modeling has not been discussed yet. It concerns *class inheritance*. Resistors, capacitances, and inductors have in common that they are all one–port elements. They all share the fact that they have two pins, each carrying a potential and a current, that the voltage drop across the one–port element can be computed as the difference between its two terminal potentials, and that current in equals current out.

It would be a pity if these common facts would have to be repeated for each model class. In terms of a Dymola notation, the generic superclass *OnePort* could be described as:

```
model class OnePort
    cut WireA(Va/i), WireB(Vb/-i)
    local u
        u = Va - Vb
end
```

Resistors and *Capacitors* could then incorporate the properties of the superclass *OnePort* into their specific definitions through a mechanism of inheritance:

```
model class Resistor
    inherit OnePort
    parameter R
        u = R*i
end
```

```
model class Capacitor
    inherit OnePort
    parameter C
        C*der(u) = i
end
```

The use of class inheritance enhances the robustness of the model, because the same code is never being manually copied and migrated to different places inside the code. Thereby, if the superclass is ever modified, the modification gets automatically migrated down into all individual model classes that inherit the superclass. Otherwise, it could happen that a user implements the modification as a local patch in one of the model classes only, being totally unaware of the fact that the same equations are also used inside other model classes.

The 3D–multibody system library supplied with Dymola makes extensive use of class inheritance in the definition of joints. *RevoluteJoint* and *PrismaticJoint* have in common that they both share the base class *OneDofJoint*. However, every *OneDofJoint* inherits the base class *Joint*.

3.3.2 Higher Index Models and Feedback Linearization

Higher index models are models with dependent storage elements. The simplest such model imaginable would be an electrical circuit with two capacitors in parallel or two inductors in series. Each capacitor or inductor is an energy storage element. However, the coupled models containing two parallel capacitors or two inductors in series still contain only one energy storage element, i.e., the coupled model is of first order only, and not of second order. Models of systems that contain algebraic equations which explicitly or implicitly relate state

variables algebraically to each other, are called *higher index models*. To be more specific, the (perturbation) index of the DAE

$$\mathbf{f}\ (\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{w}(t), t)\ =\ \mathbf{0} \tag{11}$$

is the smallest number j such that after $j-1$ differentiations of Equation (11), $\dot{\mathbf{x}}$ and $\mathbf{w}$ can be uniquely determined as functions of $\mathbf{x}$ and t. Note that $\mathbf{w}$ are purely algebraic variables, whereas $\mathbf{x}$ are variables that appear differentiated within Equation (11). Currently available DAE solvers, such as DASSL [Petzold, 1982; Brenan *et al.*, 1989], are not designed to solve DAEs with an index greater than one without modifications in the code that depend on the model structure. The reasons for this property are beyond the scope of this article (cf. [Gear, 1988; Hairer and Wanner, 1991] for details). Rather than modifying the DAE solvers such that they are able to deal with the higher index problems in a numerical fashion (which can be done, cf. e.g. [Bujakiewicz, 1994]), it may make sense to preprocess the model symbolically in such a way that the (perturbation) index of the model is reduced to one. A very general and fast algorithm for this purpose was developed by [Pantelides, 1988]. This algorithm constructs all the equations needed to express $\dot{\mathbf{x}}$ and $\mathbf{w}$ as functions of $\mathbf{x}$ and t by differentiating the necessary parts of Equation (11) sufficiently often. As a by–product, the algorithm determines in an automatic way the (structural) index of the DAE. The Pantelides algorithm has meanwhile been implemented in both Dymola and Omola.

Higher index modeling problems are closely related to *inverse models*, and in particular to *feedback linearization*, an important method for the control of nonlinear systems, cf. e.g. the article entitled "Feedback Linearization of Nonlinear Systems" by Isidori and Di Benedetto in this handbook, or [Isidori, 1989; Slotine and Li, 1991]. Inverse models arise naturally in the following control problem: given a desired plant output, what is the plant input needed to make the real plant output behave as similarly as possible to the desired plant output? If only the plant dynamics model could be inverted, i.e., its outputs treated as inputs and its inputs as outputs, solving the control problem would be trivial. Of course, this cannot usually be done, because if the plant dynamics model is strictly proper (or in the nonlinear case: exhibits integral behavior), which is frequently the case, the inverse plant dynamics model is non–proper (exhibits differential behavior). This problem can be solved by introducing a reference model with sufficiently many poles, such that the cascade model of the reference model and the inverse plant dynamics model is at least proper (doesn't exhibit differential behavior). This idea is illustrated in Fig. 5.

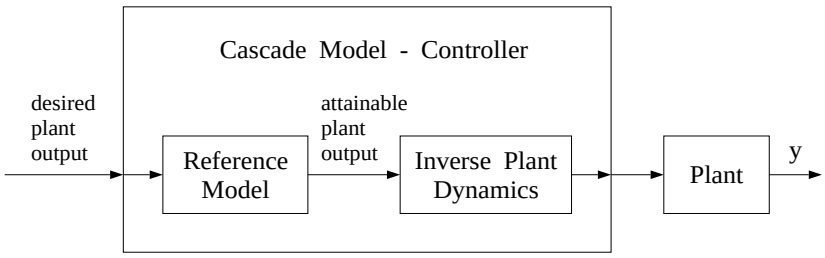

Figure 5: Control through inverse plant dynamics model.

Using the object–oriented modeling methodology, this approach to controller design can be implemented elegantly. The user would start out with the reference model and the plant dynamics model. The input of the reference model is then declared as external input, the

output of the reference model is *connected* to the output of the plant dynamics model, and the input of the plant dynamics model is declared as external output. Object–oriented modeling systems, such as Dymola, are capable of generating either a DAE or an ODE model from such a description. However, the original set of equations resulting from connecting the submodels in such a fashion is invariably of higher index. The Pantelides algorithm is used to reduce the index down to one, leading to a DAE formulation containing algebraic loops but no dependent storage elements.

Inverse dynamic models can also be used for *input–output linearization*, a special case of *feedback linearization*. The main difference to the feedforward compensation discussed above consists in using the measured state of the system instead of reconstructing this state in a separate dynamic model. To be more specific, the output equation (13) of the state–space model

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) + \mathbf{B}(\mathbf{x}) \cdot \mathbf{u} \tag{12}$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x}) \tag{13}$$

is differentiated sufficiently often, in order that the input $\mathbf{u}$ occurs in the differentiated output equations. Solving these equations for $\mathbf{u}$ allows the construction of a control law, such that the closed–loop system has purely linear dynamics. For details, cf. the article entitled "Feedback Linearization on Nonlinear Systems" by Isidori and Di Benedetto in this handbook. By interpreting Equations (12,13) as a DAE (of the type of Equation (11)), with $\mathbf{w} = \mathbf{u}$ and $\mathbf{y}$ as known functions of time, it can be noticed that the necessary differentiations to determine $\mathbf{u}$ and $\dot{\mathbf{x}}$ explicitly as functions of $\mathbf{x}$ correspond exactly to the differentiations needed to determine the index of the DAE. In other words, the Pantelides algorithm can be used to carry out this task, instead of forming the Lie brackets of Equation (13) as is usually done.

To summarize, inverse models, and in particular input–output linearization compensators, can easily be constructed by object–oriented modeling tools such as Dymola and Omola. This practical approach was described in [Mugica and Cellier, 1994].

3.3.3 Discontinuity Handling and Events

Discontinuous models play an important role in control engineering. On the one hand, control engineers often employ discontinuous control actions, e.g. when they use *bang–bang control*. However, and possibly even more importantly, the actuators that transform the control signals to corresponding power signals often contain lots of nasty discontinuities. Typically, switching power converters may exhibit hundreds if not thousands of switching events within a single control response [Glaser *et al.*, 1995].

Proper discontinuity handling in simulation has been a difficult issue all along. The problem is that the numerical integration algorithms in use are incompatible with the notion of discontinuous functions. Event detection and handling mechanisms to adequately deal with discontinuous models have been described in [Cellier, 1979]. However, many of the available modeling and simulation systems in use, such as SIMULINK, Simnon, and Desire, still don't offer appropriate event handling mechanisms. This is surprising since discontinuous models are at the heart of a large percentage of engineering problems. Only ACSL, Dymosim, SystemBuild, and some other systems offer decent event–handling capabilities.

Unfortunately, these basic event–handling capabilities are still on such a low level that it is very difficult for a user to construct a valid model of a discontinuous system even in the simplest of cases. In order to justify this surprising statement, a control circuit for the heating of a room, as shown in Fig. 6, is discussed. The heating process is described by a

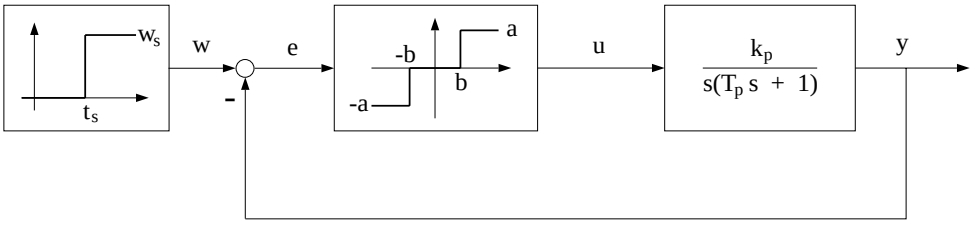

Figure 6: Simple control circuit with discontinuities.

PT1 element. The controller consists of a 3–point element together with an actuator with integral behavior, which is combined with the PT1 element in Fig. 6. At a specific time instant t_s, the set point w jumps from zero to w_s. This system can be described by the following ACSL model:

```
program HeatControl
    initial
        constant kp = 1, Tp = 0.1, a = 1, b = 0.05, ws = 1, ts = 0.5, x10 = 0, x20 = 0
        integer  mode

        ! initialize input and mode (valid for x10=0, x20=0)
            w    = 0
            mode = 0

        ! define time instant when w is jumping
            schedule setpoint .at. ts
    end

    dynamic
        derivative
            ! calculate model signals
                e = w - y
                u = mode * a
                x = integ(u, x10)
                y = kp*realpl(Tp, x, x20)

            ! define switching of 3-point controller
                schedule switch .xz. (e - b) .or. (e + b)
        end
        discrete setpoint
            w = ws
        end
        discrete switch
            if (e .lt. -b) then
                mode = -1
            elseif (e .gt. b) then
                mode = +1
            else
                mode = 0
            endif
        end
    end
end
```

During numerical integration, only the *derivative* section is executed. Since variable *mode* changes its value only when an event occurs, no discontinuity is present when integrating this section. An event occurs, when $e - b$ or $e + b$ crosses zero in either direction, or when the integration reaches time t_s. However, the above code will not always work correctly. Let us analyze some problematic situations:

1. *Initialization*:
   Before the integration starts, the *initial*, *derivative*, and *discrete* sections are evaluated once in the order of appearance. However, this does not help with the proper initialization of the variable *mode*. From the block diagram of Fig. 6, it is easy to deduce that for zero initial conditions ($x_{10} = 0$, $x_{20} = 0$) of the dynamic elements and zero input of w, the control error is zero, and therefore, *mode* has to be initialized with zero as well. However, when any of the initial conditions are non–zero, it is by no means obvious, how the variable *mode* must be initialized. A proper initialization can be done in the following way:

```
initial
    ...
    ! initialize mode
        e = w - x20
        if (e .lt. -b) then
            mode = -1
        elseif (e .gt. b) then
            mode = +1
        else
            mode = 0
        endif
end
```

   In other words, the plant dynamics must be analyzed in order to determine the correct initial value for variable *mode*. Usually this requires doubling of code of the *derivative* and *discrete* sections. This process becomes more involved as the plant grows in complexity, or when the plant itself contains discontinuous elements. Furthermore, it creates a serious barrier for modularization, because the correct initialization of a local element such as the 3–point controller, requires global analysis of the model.

   It should be noted that, even with the above initialization scheme, the simulation will be incorrect if $x_{20} = b$ and $k_p \cdot x_{10} > x_{20}$. This is due to a subtle artifact of the crossing functions. If $x_{20} = b$, $mode = 0$ and the crossing function $e + b$ is identical to zero. If $k_p \cdot x_{10} > x_{20}$, y is growing, and therefore e decreases to a value smaller than $-b$ shortly after the integration started. Since an event occurs only if a crossing function *crosses* zero, no event is scheduled. As a consequence, *mode* remains zero, although it should become -1 shortly after the integration starts. The initialization section will become even more involved if such situations are to be handled correctly.

2. *Simultaneous events*:
   What happens if a state event of the 3–point controller and the time event of the set point occur at the same time instant? In the above example, this situation can easily be provoked by simulating first with $w = 0$, determining the exact time instant at which a state event occurs and then use this time instant as initial value for t_s.

   When two events occur simultaneously, the corresponding discrete sections are executed in the order of appearance in the code. Obviously, this can lead to a wrong setting

of variable *mode*. Assuming that at the time of the event, $w = 0$ and e crosses b in the negative direction, i.e., $e = b - \epsilon$. Due to the *discrete* section *switch*, the variable *mode* will be set to zero. However, when the integration starts again, $w = w_s > 0$ and $e > b$, i.e., *mode* should be 1. In other words, *mode* has the wrong value, independently of the ordering of the discrete sections! The correct treatment of such a situation requires merging the two discrete sections into one and doubling code from the derivative section. Again, this results in a serious barrier for modularization.

It should have become clear by now that separating the modeling code into *initial*, *derivative*, and *discrete* sections, as done in ACSL, Omola, VHDL–A and other systems, is not a good idea in the presence of state events. For the user it is nearly impossible to manually generate code that is correct in all circumstances.

In [Elmqvist, *et al.*, 1993], a satisfactory solution to the problems mentioned above is proposed for *object–oriented* modeling systems, and the proposed solution has been implemented in Dymola. It is beyond the scope of this article to discuss all the details. In a nut shell, higher language elements are introduced that allow the selective activation of equations based on boolean expressions *becoming* true. These *instantaneous equations* are treated in the same way as continuous equations. In particular, they are *sorted* together with all the other equations. The sorting process automatically guarantees that the code at the initial time and at event instants is executed in the correct sequence, so that the simultaneous event problem mentioned above can no longer occur. Furthermore, the model is iterated at the initial time and after event handling to find automatically the correct switching states to prevent the initialization problem explained above from ever occurring.

To summarize, the object–oriented modeling paradigm allows a satisfactory handling of discontinuous models. This is not the case with the traditional modeling systems in use today.

3.4 Coupling of Simulation Packages

In the last section, it was discussed that modeling languages could use some sort of standardization, in order to improve the capability of simulation users to exchange models and even entire model libraries among each other. VHDL–A was mentioned as one attempt at answering this demand.

However, it had also been mentioned that more and more producers of technical equipment, such as car manufacturers, depend on second–source components *and second–source models thereof* in their system design. If every second–source provider could be forced to provide models encoded in a subset of VHDL–A, this might solve the problem. However, this will not happen for years to come. At least as an intermediate (and more easily achievable) solution, one could try to create a much lower–level standard, one for simulation run–time environments.

For efficient simulation, models have to be compiled into machine code. Portability issues suggest to generate first code in a high–level programming language, such as Fortran, C, or Ada, which is then compiled to machine code using available standard compilers. Therefore, it is natural to ask for a standardization of the *interfaces* of modeling and simulation environments at the programming language level. This allows to generate program code

from a modeling tool A, say a mechanical or electronic circuit modeling system, and use it as a component in another modeling tool B, say a block–diagram package. It is much easier to use a model at the level of a programming language with a defined interface, than writing a compiler to transform a VHDL–A model down to a programming language.

Some simulation researchers have recognized this need, and, in fact, several different low–level interface definitions are already in use:

- **DSblock** interface definition [Otter, 1992]:
  This was the first proposal for a neutral, product–independent low–level interface. It was originally specified in Fortran. The latest revision uses C as specification language, and supports the description of time–, state–, and step–event driven ordinary differential equations in state–space form, as well as regular and overdetermined DAEs of indices 1 and 2. All signal variables are characterized by text strings that are supplied through the model interface. This allows an identification of signals by their names used in the higher–level modeling environment, and not simply by an array index. Presently, Dymola generates DSblock code as interface for its own simulator, Dymosim. Also, the general–purpose multibody program SIMPACK [Rulka, 1990] can be optionally called as a DSblock.

- **SimStruct** from The MathWorks [MathWorks, 1994]:
  In the newest release of SIMULINK (Version 1.3), the interface to C–coded submodels is clearly defined, and has been named SimStruct. Furthermore, with the SIMULINK accelerator, and the SIMULINK C–Code generator, SIMULINK can generate a SimStruct model from a SIMULINK model consisting of any built–in SIMULINK elements and from SimStruct blocks (S–functions written in the MATLAB language cannot yet be compiled). A SimStruct block allows the description of input/output blocks in state–space form consisting of continuous– and discrete–time blocks, with multi–rate sampling of the discrete blocks[3]. However, neither DAEs nor state–events are supported. DAEs are needed in order to allow the incorporation of model code from domain specific modeling tools like electric circuits or mechanical systems. State–events are needed in order to properly describe discontinuous modeling elements and variable structure systems.

- **User Code Block (UCB)** interface from Integrated Systems [Integ. Systems, 1994]:
  The UCB–interface used with SystemBuild allows the description of time– and state–event dependent ordinary differential equations in state–space form, as well as regular and overdetermined DAEs of index 1. It is more general than the SimStruct interface. Some commercial multibody packages (e.g. SIMPACK [Rulka, 1990], DADS [Smith and Haug, 1990]) already support this interface, i.e., can be used within SystemBuild as an input/output block. Two serious drawbacks are still present in this definition. First, the dimensions of model blocks have to be defined in the SystemBuild environment. This means that model blocks from other modeling environments, such as mechanical and electrical systems, cannot be incorporated in a fully automated fashion, because the system dimensions depend on the specific model components. Contrarily, in the DSblock interface definition, the model dimensions are reported

[3]Multi–rate sampling is a special case of time–events.

from the DSblock to the calling environment. Second, variables are identified by index in the SystemBuild environment. This restricts the practical use of the tool to models of low to medium complexity only.

4 Shortcomings of Current Simulation Software

As already discussed, a serious shortcoming of *most* simulation tools currently on the market is their inability to treat discontinuous models adequately. This is critical because most real–life engineering problems contain discontinuous components. Sure, a work–around for this type of problem consists in modeling the system in such a detail that no discontinuities are present any longer. This is e.g. done in the standard electric circuit simulator SPICE. However, the simulation time increases then by a factor of 10–100, and this limits the size of systems that can be handled economically. Note that proper handling of discontinuous elements is *not* accomplished by just supplying language elements to handle state events, as is done e.g. in ACSL or SystemBuild. The real problem has to do with determining the correct mode the discontinuous system is in at all times. Object–oriented modeling can provide an answer to this critical problem.

Block–diagram editors are, in the view of the authors of this article, a cul–de–sac. They *look* modern and attractive, because they employ modern graphical input/output technology. However, the underlying concept is unnecessarily and unjustifiably limited. Although it is trivial to offer block–diagram editing as a special case within a general–purpose object–diagram editor, the extension of block–diagram editors to object–diagram editors is far from trivial. It is easy to predict that block–diagram editors will be replaced by object–diagram editors in the future, in the same way as block–diagram editors have replaced the textual input of general–purpose simulation languages in the past. However, it may take several years before this will be accomplished. Most software vendors only react to pressure from their customers. It may still take a little while before sufficiently many simulation users tell their simulation software providers that object–diagram editors is what they need and want.

Although the introduction of block–diagram simulators has enhanced the user–friendliness of simulation environments a lot, there is still the problem with model component libraries. As previously explained, the "library" technique supported by block–diagram simulation systems, such as SIMULINK and SystemBuild, is only of limited use, because a modification in a component in a library cannot easily be incorporated into a model in which this component is being used. Again, the object–oriented modeling systems together with their object–diagram editors provide a much better and more satisfactory solution.

Beside from the modeling and simulation issues discussed in some detail in this article, there exists the serious practical problem of organizing and documenting simulation experiments. To organize the storage of the results of many simulation runs, possibly performed by different people, and to keep all the information about the simulation runs necessary in order to reproduce these runs in the future, i.e., store the precise conditions under which the results have been produced, is a problem closely related to version control in general software development. At present, nearly no support is provided for such tasks by available simulation systems.

5 Conclusions

The survey presented in this article is necessarily somewhat subjective. There exist several hundred simulation software packages currently on the market. It is evident that no single person can have a working knowledge of all these packages. Furthermore, software is a very dynamic field that is constantly enhanced and upgraded. The information provided here represents our knowledge as of July 1995. It may well be that some of our criticism will already be outdated by the time the reader lays his eyes on this handbook.

To summarize, the textual simulation languages of the past have already been largely replaced by block–diagram editors, since these programs are much easier to use. Most simulation programs entered through block–diagram editors still have problems with efficiency, because equations are interpreted rather than compiled into machine code. For larger systems, the right choice of the simulation package is therefore still not easy. The future belongs definitely to the object–oriented modeling languages and their object–diagram editors, since these new techniques are much more powerful, reflect more closely the physical reality they try to capture, and contain the popular block–diagrams as a special case.

Acknowledgments

The authors would like to acknowledge the valuable discussions held with and comments received from Hilding Elmqvist and Ingrid Bausch–Gall.

References

Andersson, M. 1990. *Omola — An Object–Oriented Language for Model Representation*, Licenciate thesis TFRT–3208, Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Andersson, M. 1992. Discrete event modelling and simulation in Omola. *Proc. IEEE Symp. Computer–Aided Control System Design*, Napa, CA, pp. 262–268.

Andersson, M. 1994. *Object–Oriented Modeling and Simulation of Hybrid Systems*, Ph.D. thesis TFRT–1043, Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Augustin, D.C., Fineberg, M.S., Johnson, B.B., Linebarger, R.N., Sansom, F.J., and Strauss, J.C. 1967. The SCi Continuous System Simulation Language (CSSL). *Simulation*, 9:281–303.

Brenan, K.E., Campbell, S.L., and Petzold, L.R. 1989. *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations*, Elsevier Science Publishers, New York, new ed. to appear in 1995.

Bujakiewicz, P. 1994. *Maximum Weighted Matching for High Index Differential Algebraic Equations*, Ph.D. thesis, Technical University Delft, The Netherlands.

Cellier, F.E. 1979. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D. dissertation, Diss ETH No 6483, ETH Zurich, Zurich, Switzerland.

Cellier, F.E. 1991. *Continuous System Modeling*, Springer–Verlag, New York.

Cellier, F.E. 1993. Integrated continuous–system modeling and simulation environments. In *CAD for Control Systems*, ed. D.A. Linkens, p. 1–29, Marcel Dekker, New York.

Elmqvist, H. 1975. *Simnon — An Interactive Simulation Program for Nonlinear Systems*, Report CODEN:LUTFD2/(TFRT–7502), Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Elmqvist, H. 1978. *A Structured Model Language for Large Continuous Systems*, Ph.D. dissertation. Report CODEN:LUTFD2/(TFRT–1015), Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden.

Elmqvist, H. 1995. *Dymola — User's Manual*, Dynasim AB, Research Park Ideon, Lund, Sweden.

Elmqvist, H., Åström, K.J., Schönthal, T., and Wittenmark, B. 1990. *Simnon — User's Guide for MS-DOS Computers*, SSPA Systems, Gothenburg, Sweden.

Elmqvist, H., Cellier, F.E., and Otter, M. 1993. Object–oriented modeling of hybrid systems. *Proc. ESS'93, European Simulation Symp.*, Delft, The Netherlands, pp. xxxi–xli.

Gear, C.W. 1988. Differential–algebraic equation index transformations. *SIAM J. Scientific and Statistical Computing*, 9:39–47.

Glaser, J.S., Cellier, F.E., and Witulski, A.F. 1995. Object–oriented power system modeling using the Dymola modeling language. *Proc. Power Electronics Specialists Conf.*, Atlanta, GA, vol. II, pp. 837–843.

Hairer, E. and Wanner, G. 1991. *Solving Ordinary Differential Equations II. Stiff and Differential–Algebraic Problems*, Springer–Verlag, Berlin.

Integrated Systems Inc. 1994. *SystemBuild User's Guide*, Version 4.0, Santa Clara, CA.

Isidori, A. 1989. *Nonlinear Control Systems: An Introduction*, Springer–Verlag, Berlin.

Korn, G.A. 1989. *Interactive Dynamic–System Simulation*, McGraw–Hill, New York.

Ljung, L. 1987. *System Identification*, Prentice–Hall, Englewood Cliffs, N.J.

Mathworks Inc. 1992. *SIMULINK — User's Manual*, South Natick, MA.

Mathworks Inc. 1994. *SIMULINK — Release Notes Version 1.3*, South Natick, MA.

Mitchell and Gauthier Assoc. 1991. *ACSL: Advanced Continuous Simulation Language — Reference Manual*, 10th ed., Mitchell & Gauthier Assoc., Concord, MA.

Mugica, F. and Cellier, F.E. 1994. Automated synthesis of a fuzzy controller for cargo ship steering by means of qualitative simulation. *Proc. ESM'94, European Simulation MultiConference*, Barcelona, Spain, pp. 523–528.

Otter, M. 1992. *DSblock: A Neutral Description of Dynamic Systems*, Version 3.2. Technical Report TR R81–92, DLR, Institute for Robotics and System Dynamics, Wessling, Germany. Newest version available via anonymous ftp from "rlg15.df.op.dlr.de" (129.247.181.65) in directory "pub/dsblock".

Pantelides, C.C. 1988. The consistent initialization of differential–algebraic systems. *SIAM J. Scientific and Statistical Computing*, 9:213–231.

Petzold, L.R. 1982. A description of DASSL: A differential/algebraic system solver. *Proc. 10th IMACS World Congress*, Montreal, Canada.

Rodrigues, J. 1994. 1994 Directory of simulation software. SCS — The Society For Computer Simulation, Vol. 5, ISBN 1-56555-064-1.

Rulka, W. 1990. SIMPACK — a computer program for simulation of large–motion multibody systems. In *Multibody Systems Handbook*, ed. W. Schiehlen, Springer–Verlag, Berlin.

Slotine, J.-J. E. and Li, W. 1991. *Applied Nonlinear Control*, Prentice–Hall, Englewood Cliffs, N.J.

Smith, R.C. and Haug, E.J. 1990. DADS — Dynamic Analysis and Design System. In *Multibody Systems Handbook*, ed. W. Schiehlen, Springer–Verlag, Berlin.

Vachoux, A. and Nolan, K. 1993. Analog and mixed–level simulation with implications to VHDL. *Proc. NATO/ASI Fundamentals and Standards in Hardware Description Languages*, Kluwer Academic Publishers, Amsterdam, The Netherlands.

Vachoux, A. 1995. VHDL–A archive site (IEEE DASC 1076.1 Working Group) on the Internet on machine "nestor.epfl.ch" under directories "pub/vhdl/standards/ieee/1076.1" to get to the read–only archive site, and "incoming/vhdl" to upload files.