GUIDELINES FOR MODELING AND SIMULATION OF HYBRID SYSTEMS

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

Hilding Elmqvist, DynaSim AB, Research Park Ideon, 223 70 Lund, Sweden, e_mail: Elmqvist@Gemini.LDC.LU.SE

Martin Otter, Department of Robotics and System Dynamics, German Aerospace Research Establishment Oberpfaffenhofen, D–8031 WESSLING, Germany, e_mail: DF43@VM.OP.DLR.DE

James H. Taylor, ORA Corporation, 301 Dates Drive, Ithaca, New York 14850–1313, U.S.A, e_mail: Jim@ORACorp.Com

Abstract. Hybrid systems are defined as systems that exhibit mixed properties of continuous–time systems, discrete–time systems, and discrete–event systems. The paper analyzes various aspects that need to be taken into consideration when modeling such systems for the purpose of a numerical simulation. It addresses issues related to the convenience of the modeler when dealing with such systems, as well as issues related to the numerical properties of the generated simulation code. It shows that the difficulties encountered in modeling such systems are quite different in nature from those encountered in simulating them. When addressed in their proper contexts, these two types of problems can almost totally be decoupled from each other. A set of guidelines for the design of hybrid modeling and simulation software is derived that may support future software designers in their attempt at developing products that are both more user–friendly in terms of their modeling capabilities and more robust in terms of their simulation capabilities.

Keywords. Hybrid systems; discontinuity handling; modeling software; simulation software; software standards.

DISCONTINUITY HANDLING AND STEP–SIZE CONTROL

The Continuous–System Simulation Language (CSSL) standard of 1967, on which most of today's simulation languages are based, does not discuss discontinuities in models at all (Augustin *et al.*, 1967). When the document was written, the capabilities of the digital computers available at that time were still so limited both in terms of speed and memory that complex engineering models had little chance of being handled by them in any meaningful way. Consequently, more complex models were still run on analog computers where simple discontinuities in equations don't pose problems.

Taking this evident oversight of the CSSL committee into account, it is surprising to realize that the CSSL standard has nevertheless survived for a quarter of a century, and languages designed and developed following this standard are today employed to describe models that are orders of magnitude more complex than anything the standard had originally been designed for. Many of the problems tackled today by means of simulation are discontinuous in nature. Mechanical systems, such as robots, are plagued by dry friction, backlash, and hysteresis, sudden contact between bodies, as well as discontinuous input trajectories. Digital electronic circuits are characterized by switching devices such as switching MOSFETs and Zener diodes, as well as discontinuous time–dependent source elements. The behavior of plants from the chemical process industry is largely dictated by opening and closing valves, by heating elements that are turned on or off, and by batches of new raw material being added somewhere in the process at distinct points in time. All these effects result in discontinuous plant behavior of some sort.

Part of the evident success of the CSSL standard is due to the fact that the numerical integration algorithm, or more precisely its step–size control component, assists the simulation software in dealing with discontinuities. This fact can be explained easily. All numerical integration algorithms employed in today's simulation software operate on the basis of Taylor series approximations. Different integration algorithms vary in how they approximate the higher state derivatives and in the number of terms of the Taylor series expansion that they consider in the approximation (the so–called order of the integration algorithm).

Step–size control is frequently based on the assumption that the difference in the results obtained when repeating the same integration step twice with two different integration algorithms is proportional to the actual, yet unknown, error made in the numerical approximation using either of the two techniques. This would work if one had two matched algorithms, one of which always overestimates the analytical solution while the other always underestimates it, yet such that both algorithms approach the analytical solution smoothly as the step size, h, approaches zero.

Unfortunately, such a pair of matched algorithms cannot exist. The higher–order terms of the Taylor series expansion become less and less important as the step size is reduced. Consequently, all integration algorithms will behave like an Euler (first–order) algorithm for sufficiently small step sizes. Thus, by reducing the step size, it is always possible to get the two algorithms to agree. However, they won't agree on the analytical solution; instead, they will agree on a first–order approximation.

When a discontinuity takes place within the integration step, the Taylor–Series approximation will invariably be inaccurate since polynomials don't exhibit discontinu-

ities. Consequently, the two algorithms will disagree on the outcome, which triggers the step–size control algorithm to reduce the step size, and this somewhat helps to locate the discontinuity. However, in the process, the step size is being reduced until, eventually, the two integration algorithms agree on the same, yet still incorrect, result since both behave like Euler. This often leads to "creeping" effects (integration with very small step size), and may possibly result in completely incorrect answers. This was shown in Cellier (1986).

It is important to distinguish between different types of discontinuities. A function $f(x)$ is called *discontinuous* if there exists at least one point, x^*, where:

$$\lim_{x \to x^*} f(x) \neq \lim_{x^* \leftarrow x} f(x) \tag{1}$$

A function for which no such point can be found is called *continuous*. A continuous function is said to have a *discontinuous derivative* if the function $g(x) = \partial f(x)/\partial x$ is discontinuous. A function with a continuous derivative has a *discontinuous second derivative* if the function $h(x) = \partial g(x)/\partial x$ is discontinuous, etc. Since no generally accepted short names for these types of functions have been introduced in the technical literature so far, the remainder of this paper shall refer to discontinuous functions as *D–functions*, to functions with a discontinuous first derivative as *DD–functions*, etc.

From a numerical perspective, D–functions pose the most severe problems, since even the Euler integration algorithm cannot deal with them. Imprudent integration across discontinuities in D–functions will frequently lead to simulation results that are entirely incorrect. DD–functions are less malignant since the discontinuity now shows up only in the second state derivative. The Euler algorithm will usually be able to handle such situations since it does not require an estimate of the second state derivative. Simulations involving DD–functions will slow down in the vicinity of the discontinuities since the integration algorithm has to drop down to first order by using a very small step size to clear the discontinuity, but at least, the results are most likely to be correct (although not necessarily, since the thyristor example mentioned in Cellier (1986) only contained DD–functions). Models involving DDD–functions will slow down less since the integration algorithm only needs to drop down to second order, etc.

DISCONTINUITY SMOOTHING AND NUMERICAL INTEGRATION

A remedy that is often employed by engineers is to smooth out the discontinuity. In reality, the discontinuity may not be as hard as the ideal model presumes, and thus, it may be quite justifiable for the model to represent the discontinuity as a continuous curve with a locally large gradient. The smoothing can be accomplished either statically by smoothing out the discontinuous function directly, or dynamically by introducing an additional very fast time constant. The latter approach is commonly used to circumvent algebraic loops in model equations.

While this approach does away with the theoretical problems of discontinuity handling, it isn't really helpful from a numerical perspective. Numerically, there is no difference between a discontinuity and a locally large gradient. If an explicit Runge–Kutta algorithm is used to integrate the state–space model, its step size will have to be reduced in the vicinity of the large gradient in order to keep the solution numerically stable, since the local Jacobian will temporarily have an eigenvalue far out along the negative real axis. That is, in the vicinity of points of high curvature, very small step sizes are multiplied with very large and inaccurately estimated higher–order derivatives, which is numerically questionable. On the other hand, if a stiffly–stable algorithm is used (most simulation languages offer a BDF method (Gear, 1972) for this purpose) to avoid the need to reduce the step size in order to keep the solution numerically stable, the results will be even worse since BDF algorithms are numerically awful when applied to a problem with fast changing eigenvalues of its Jacobian. Consequently, discontinuity smoothing and the introduction of additional fast eigenvalues are techniques that don't work very well in practice.

EVENT SCHEDULING AND DISCONTINUITY ITERATION

From the previous exposé, it has become evident that neither step–size adjustment nor smoothing solve the problems posed by discontinuities in the system. Adding more details to the model to cover up its inherently discontinuous behavior won't help with the numerics. On the contrary, an additional step toward abstraction should be taken.

Many of the above issues are illustrated by considering the limiter function of Fig. 1:

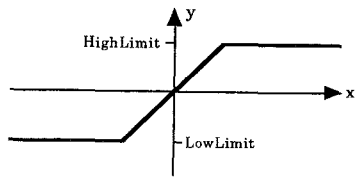

Figure 1. Limiter function.

it can be noticed that this DD–function consists of three perfectly continuous domains connected by point discontinuities in their first derivative. Since discontinuities impair the numerical integration process as outlined above, the integrator should be prevented from passing through any discontinuity. To this end, the three domains are extended in a continuous fashion, as indicated in Fig. 2.

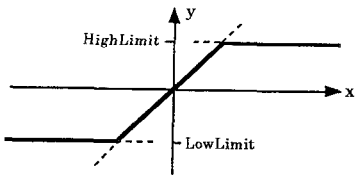

Figure 2. Extended limiter function.

During the process of numerical integration, only one of the three domains can be active at any one time. Demons are installed that watch over the boundaries of the domains. When a trajectory tries to enter an extended area, the associated demon awakes and initiates an iteration procedure to locate the boundary accurately. Once the boundary has been located, the numerical integration is suspended, and the discontinuity (a so–called state event) is processed, which switches from one domain to the next. Once the event has been processed, numerical integration can be resumed, however, the integration algorithm must be restarted afresh. This process was outlined in Cellier (1979).

Some designers of CSSL–type languages have added event scheduling mechanisms to their products. Most prominently among the CSSL languages that offer event handling is ACSL (Mitchell & Gauthier, 1991). In ACSL, the limiter function can, for example, be programmed as follows:

```
PROGRAM LimiterTest
        constant HighLimit = 1.0, LowLimit = -1.0
        logical High, Low

    INITIAL
        x = 2 * sin(t)
        High = x .ge. HighLimit
        Low = LowLimit .ge. x
    END ! of INITIAL

    DYNAMIC
        DERIVATIVE
            x = 2 * sin(t)

            if (High) then y = HighLimit
            elseif (Low) then y = LowLimit
            else y = x end if

            schedule LimitEvent / SetHigh .xp. x - HighLimit
            schedule LimitEvent / ResetHigh .xn. x - HighLimit
            schedule LimitEvent / SetLow .xp. LowLimit - x
            schedule LimitEvent / ResetLow .xn. LowLimit - x
        END ! of DERIVATIVE

        DISCRETE LimitEvent
            High = SetHigh .or. High .and. .not. ResetHigh
            Low = SetLow .or. Low .and. .not. ResetLow
        END ! of DISCRETE LimitEvent

        termt t .ge. 10.0
    END ! of DYNAMIC
END ! of PROGRAM
```

The Boolean variables *High* and *Low* are used to keep track of which of the three piecewise continuous curves of Fig.2 is currently active. The value of these variables never changes while the numerical integration is in progress. The *schedule* statements are the demons that watch over the execution of the continuous simulation model. When $x - HighLimit$ crosses 0.0 in positive direction (*.xp.*), the state event *LimitEvent* is scheduled to be executed. As soon as this demon awakes, an iteration process starts to locate the zero crossing exactly, while the model continues to operate in the center state with *High* = *.false.* on both sides of the discontinuity. After the iteration has converged, the continuous simulation is suspended, the Boolean variable *SetHigh* named in the *schedule* statement is set to *.true.*, and the discrete section is executed once. This block will set *High* to *.true.* according to the first Boolean assignment, after which integration can resume.

Special statements have been added to initialize *High* and *Low* in accordance with the initial value of x, so that they carry the correct value before being used to compute the initial value of y.

ACSL's notation is a little clumsy. A more compact, readable, and equivalent notation will be presented later. However, from a numerical point of view, ACSL's solution usually works well. A more recent effort (Taylor, 1993) has extended the Simnon language ((Elmqvist *et al.*, 1990; Frederick and Taylor, 1989) to provide the same functionality that ACSL offers with a number of advantages.

Notice that the above outlined approach will not always work. Figure 3 shows the turbulent hydraulic resistance across a valve. The volume flow rate q is computed as a function of the pressure difference Δp.

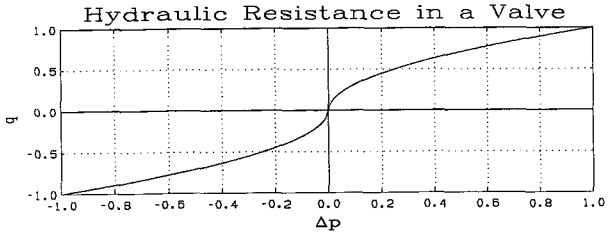

Figure 3. Hydraulic resistance across a valve.

This function is of the DDD-type, and it may therefore not be crucial to locate the discontinuity in the second derivative. It may in fact be quite dangerous to do so since, at the place where the second derivative is discontinuous, the first derivative happens to be infinitely large.

Trying to apply the previously advocated technique would suggest use of the formula:

$$q = k \cdot \sqrt{\Delta p} \tag{2a}$$

for $\Delta p \geq 0.0$, and:

$$q = -k \cdot \sqrt{-\Delta p} \tag{2b}$$

for $\Delta p < 0.0$. Unfortunately, neither of these formulas will work since each of the two terms is defined in $\Re^1$ only within its own range. The two formulas don't extends naturally and continuously into the other domain. Since the overall function is in fact only of the DDD-type, although Eq.(1) suggests two separate branches, it may make more sense to program the function directly, i.e., without any event scheduling, using the formula:

$$q = k \cdot \text{sign}(\Delta p) \cdot \sqrt{|\Delta p|} \tag{3}$$

A worse situation is plotted in Fig. 4.

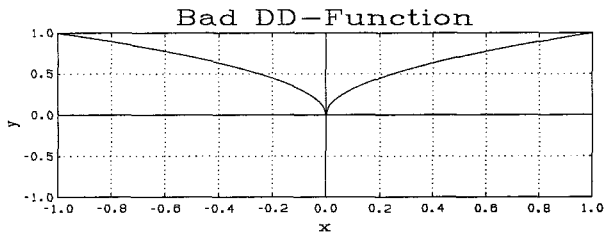

Figure 4. Bad DD-function.

In this case, a combination of the two previous approaches may work. The two domains could be extended using an enhanced version of Eq.(3):

$$y = m \cdot k \cdot \text{sign}(x) \cdot \sqrt{|x|} \tag{4}$$

where m is a mode parameter with a value of $+1.0$ for $x \geq 0.0$ and -1.0 for $x < 0.0$. Unfortunately, the zero crossing must now be located accurately, and if the partial derivative of this function turns out to be important, this may again not work properly. Fortunately, this utterly malignant function is not something that physics often prescribes.

HYBRID OBJECT–ORIENTED MODELING

Up to this point, the focus of the paper has been on the numerical properties of hybrid system simulation. From here onward, the paper will focus on the convenience of hybrid system modeling. As previously explained, the mechanism offered in ACSL to describe hybrid systems is quite acceptable from a numerical perspective although a little clumsy from a modeling point–of–view. Unfortunately, what seemed to be a minor inconvenience in this simple example becomes a major hurdle when modeling large–scale industrial processes. The problems encountered are twofold:

1. While modeling, the user must constantly be aware of the numerical procedure that is employed to simulate his or her model. This fact was illustrated by means of the three different notations needed to tackle the different types of discontinuities. The primary purpose of modeling software should be to protect the user from a detailed knowledge of the intricacies of the underlying numerical algorithms. The user should be enabled to

concentrate on issues related to the physics behind his or her model, rather than being forced to spend much time on thinking about what the simulator will do with the model once it has been designed and encoded.

2. The topological structure of the model should reflect the topological structure of the underlying physical process being modeled. This is a key issue in successful modeling of large–scale industrial processes. This concept is commonly referred to as the object–oriented programming paradigm (Cellier *et al.*, 1991). Today's CSSL–type languages are not object–oriented. They provide for a structuring capability, called a *macro*. However, macros are not truly modular as shown in Cellier and Elmqvist (1993). ACSL's approach to hybrid system modeling forces the user to encode three aspects of one and the same physical phenomenon (the demon, the event handler, and the continuous–time equation) in three different places of the model. This leads to poorly maintainable code when the model contains not one but maybe several dozens of discontinuous functions.

Reflecting on the two difficulties outlined above, the reader may remember an earlier remark: Discontinuity handling did not pose any major problems in the days of analog computing. Why was that? Switching functions were simply implemented by comparator elements, and switching took place whenever the time for it had come. There was no need to iterate to locate the precise time of an event, *since the analog computer programmer could rely on time being a continuously advancing variable.*

The difficulties all began with the realization that, in a digital simulation program, time jumps forward in discrete steps. Thereby it can (and will) happen that event times are missed, which creates the need for demons to discover such mishap and for iteration algorithms to recover from it. Worse, some integration algorithms don't even guarantee that *Time*, during numerical integration, always proceeds forward. Integration algorithms with error control and variable step size let *Time* jump forward and backward during the integration across each time–step. It is thus *not* possible to write a statement like:

```
if (Height < 0.0) then
    Velocity = -c * Velocity
end if
```

in ACSL or in a Fortran subroutine calulating derivatives to make a ball bounce when it reaches the floor. The integrator might step back to a situation with $Height > 0.0$, but then the *Velocity* has already changed its sign.

If only the modeler could rely on *Time* being a continuously advancing variable, most of his or her problems would vanish at once. Thus, the answer is simple. The authors of this paper decree that, in a hybrid modeling language, *Time* may be perceived as a continuously advancing variable. This ordinance relieves the modeler of the need to think about the numerical implications of formulating discontinuous models, and puts instead the burden on the designer of the modeling language compiler to interpret the model in a way as to generate code for the underlying simulation language such that it can be executed in a numerically sound fashion. What does not work is that *if* statements like the one above are mapped into *if* statements of the underlying simulation language, but this does not have to happen. It is perfectly feasible for a smart modeling language compiler to interpret the above *if* statement, realize that it implies a discontinuous function, and synthesize the necessary *schedule* statements and discrete events for the underlying ACSL program to ensure a numerically sound execution of the generated simulation code. Alternatively, crossing functions and event subroutines can be generated in Fortran, for example, in accordance with the DSblock specification (Otter, 1992). It will be shown that this simple recipe puts to the sword once and for all the need for splitting descriptions of discontinuous functions into several disjoint program segments. It thereby supports true object–oriented modeling.

High–level notations will now be discussed that make it much easier for the modeler to describe discontinuities and events. Dymola is a language that supports these high–level abstractions.

Dymola is a modeling language that systematically builds on the object–oriented programming paradigm (Elmqvist, 1978; Cellier, 1991; Cellier and Elmqvist, 1993; DynaSim, 1993). Dymola models are truly modular and can be hierarchically composed in a manner that allows to represent the topological structure of a physical system faithfully. Dymola is a code generator that can generate code for various simulation languages, such as ACSL (Mitchell & Gauthier, 1991), DESIRE (Korn, 1989), Simnon (Elmqvist *et al.*, 1990; Frederick and Taylor, 1989), and SIMULINK (MathWorks, 1992). Furthermore, it can generate code directly in plain Fortran using either the Simnon– or the DSblock–format (Otter, 1992).

The previously mentioned limiter can be specified in Dymola using *if* expressions as follows:

$$\begin{aligned} y = &\textbf{if } x > HighLimit \textbf{ then } HighLimit \\ &\textbf{elseif } x < LowLimit \textbf{ then } LowLimit \textbf{ else } x \end{aligned} \tag{5}$$

Note the difference between *if expressions*, which always return one value, and *if statements*.

This description is considerably more compact and more convenient to write down than the previously introduced ACSL code for the same purpose. Yet, the Dymola compiler will recognize the implied state events, and will automatically generate appropriate *schedule* statements and *discrete* sections when asked to synthesize an ACSL program. The result is approximately the same as the previously shown ACSL program. However, Dymola employs a more general method for variable initialization, and, of course, the Dymola compiler wouldn't know how to synthesize semantically meaningful variable names such as *HighLimit* or *ResetLow*.

In this way, the modeler can preserve the best of two worlds: compact, easy–to–write, easily readable and maintainable, object–oriented model code at the level of the modeling language; yet, properly executing, numerically sound, run–time efficient simulation code at the level of the simulation language.

It is important to realize that these are two separate issues that can be decoupled from each other both conceptually and practically. Object–oriented modeling is the way to go, yet, object–oriented simulation is quite problematic (Cellier *et al.*, 1991). The *model language* code should support a structuring of the model that reflects the topology of the underlying physical process, but should be monolothic with respect to the numerical necessities of the resulting simulation program (things that logically belong together must be describable as one piece of model language code). On the other hand, the *simulation* language code should be monolithic with respect to the structure of the underlying physical processes (for reasons of run–time efficiency), but must be appropriately structured to support the numerical algorithms that ultimately execute the synthesized simulation program.

INSTANTANEOUS EQUATIONS

Up to this point, the discussion has focused on events that occur as an indirect consequence of discontinuities in functions or function derivatives. However, this view is too limited. Mechanisms are also needed to describe sudden changes in the model structure, changes that cannot be properly reflected by merely altering the expression that assigns a value to a state derivative or an algebraic model variable. Such mechanisms are needed, for example, to describe what happens at the boundaries of a hysteresis function, or to model computer algorithms that are used as part of sampled data systems. Instantaneous equations are also used to describe changes to continuous states that occur only at particular instants of time, e.g., to model the behavior of a bouncing ball at impact.

An *instantaneous equation* takes the form:

```
when < condition > then
    < equations >
end when
```

The equations are valid only at the instant when the condition *becomes* true.

For example, a hysteresis function can be modeled by:

```
when u > H or u < -H then
    y = if u > H then 1 else -1
end when
```

The output y changes its value only when u *becomes* greater than H or *becomes* smaller than $-H$.

Difference equations are valid at certain time instants. They define *discrete variables* that usually are functions of their own past. A previous value of a discrete state x is referred to as $old(x)$.

A difference equation and the time instants when it is valid can thus be described as:

```
when Time >= old(NextTime) then
    y + a*old(y) = b * u
    NextTime = Time + SamplingRate
end when
```

It is also possible to change continuous state variables when a certain condition is met (cf. the previous erroneous use of an *if* statement):

```
when Height < 0.0 then
    new(Velocity) = -c * Velocity
end when
```

It is essential to be able to propagate and synchronize events. Boolean variables and Boolean equations are used for this purpose. Boolean variables change their values due to some relation becoming *true* or *false* or indirectly due to some instantaneous equation.

The translation of instantaneous equations is performed with an algorithm similar to the one used for *if* expressions. The relations in the condition of the *when* equation thus are also translated into crossing functions.

THE WARD–LEONARD SYSTEM: AN EXAMPLE

A Ward–Leonard drive consists of three rotating machines, two of which are used as motors, one as a generator. The primary motor drives the generator with constant speed. The generator, in turn, is used as a constant source of current making the angular velocity of the secondary motor less load–dependent. The Ward–Leonard drive is shown on Fig. 5.

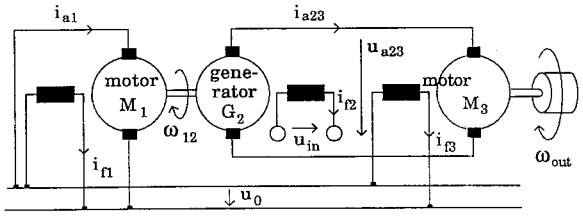

Figure 5. Ward–Leonard drive.

It is assumed that the three machines are DC–motors. Friction effects of the bearings and suspensions are described using a realistic friction model as shown on Fig. 6.

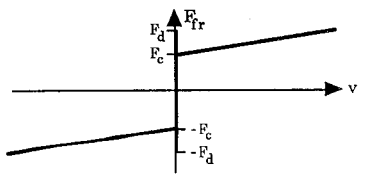

Figure 6. Friction model.

The Dymola code needed to describe the friction model is as follows:

```
model type friction

    terminal v, Fm, Ffr
    local Forward =false, Backward =false, Moving
    parameter Fd = 5.0E-5, Fc = 2.0E-5, Rv = 2.0E-4

        Moving   = old(Forward) or old(Backward)
        Forward  = not Moving and Fm > Fd
                   or old(Forward) and v > 0.0
        Backward = not Moving and Fm <-Fd
                   or old(Backward) and v < 0.0

        Ffr =      if Forward then Rv * v + Fc
                   else if Backward then Rv * v - Fc
                   else Fm

end
```

The friction phenomenon is described by a continuously changing algebraic variable, *Ffr*, which, similarly to the previously described *limit* function, switches its model in accordance with implicitly formulated state events. Since the switching conditions are much more complex in this example, they are not described inside the *if* expression, but are pulled out as boolean expressions. Thereby, the discrete state variables that are used to select the correct branch of the model are made explicit, whereas they were implicized in the much simpler *limit* function.

Two boolean variables are used in the model, called *Forward* and *Backward*. They are discrete state variables, simultaneously assuming the role of mode selectors. The third mode selector, *not Moving*, is a function (*not*) of a discrete algebraic variable (*Moving*). *Forward* becomes true if the current mode is *not Moving* and if the pulling force becomes sufficiently strong, and it becomes false if the current mode is *Forward* and the velocity, v, becomes negative. Each time the discrete state variable *Forward* changes its value, a state event is triggered, since *Forward* is used as a condition in the *if* expression to compute *Ffr*.

The reader is invited to compare this compact description to the elaborate description of friction modeling given in the ACSL manual (Mitchell & Gauthier, 1991). Yet, the process of translating this compact description into an equivalent ACSL formulation is completely systematic and can be fully automated.

At this point, the motor can be described.

```
model type dcmot

    submodel (friction) Fr

    cut arma(ua/ia), field(uf/if), mech(tau/-omega)
    main path P < arma - mech >
    local ui, psi, taum, tauR, tauFr
    parameter Ra = 10.0, Rf = 25.0, Jm = 0.05, kmot = 0.5

        Fr.v = omega
        Fr.Fm = tauR
        tauFr = Fr.Ffr

        uf = Rf * if
        ua = ui + Ra * ia
        psi = kmot * if
        taum = psi * ia
        ui = psi * omega
        tauR = taum - Jm * der(omega)
        tau = tauR - tauFr

end
```

The DC–motor model invokes a submodel of type *friction* called *Fr*. It is plugged into the model by explicit assignment of variables using a (Pascal–like) dot–notation. *Fr.vm* denotes the *vm* variable of model *Fr*. For simplicity, the armature and field inductances were omitted.

A constant voltage source is also needed. Its model is given below.

```
model type VSource
    main cut A(u/-i)
    parameter U0 = 1.0
        u = U0
end
```

The load consists of a torque load and an inertial load:

```
model type Load
    cut in(tau/omega), out(theta)
    main path P < in - out >
    terminal Jl, taul
        tau = Jl * der(omega) + taul
        der(theta) = omega
end
```

At this point, the Ward–Leonard drive can be assembled.

```
model type WLdrive

    submodel (VSource) U0(U0 = 25.0)
    submodel (dcmot) Mot1, Gen, Mot2
    submodel (Load) Ld

    cut in(u/i), out(theta, omega, tau)
    main path P < in - out >
    terminal Jl, taul

        connect Mot1 to (reversed Gen) to Mot2 to Ld
        connect U0 at Mot1 : arma at Mot1 : field at Mot2 : field
        connect in at Gen : field

        Ld.Jl = Jl
        Ld.taul = taul
        theta = Ld.theta
        omega = Ld.omega
        tau = Mot2.tau

end
```

The reader is encouraged to compare this model with the schematic of Fig. 5. This time, more advanced connection mechanisms have been used. Dymola's *connect* statement allows to either connect entire *cuts* using the *at* construct, or entire *paths* using the *to* construct. Thereby bundles of variables are connected at once. Variables to the left of the "/" separator are across variables (like voltages of electrical circuits), whereas variables to the right of the "/" separator ar through variables (like currents in electrical circuits). During compilation, *connect* statements are automatically expanded to sets of individual equations.

At this point, it is important to rethink the concept of state variables. Obviously, the angular velocities of *Mot1* and *Gen* are identical, and therefore, they don't both qualify as state variables of the generated simulation program. Yet, they are "state variables" of the individual models (meaning that they are indirectly defined by means of the *der(omega)* construct). Consequently, the Ward–Leonard drive constitutes a so–called higher–index model (Brenan *et al.*, 1989). The overall system order is lower than the sum of the orders of its submodels. Dymola employs the Pantelides algorithm (Pantelides, 1988) to automatically reduce higher–index models to state–space form (Cellier and Elmqvist, 1993).

A position control circuit involving the previously defined Ward–Leonard drive is shown on Fig. 7.

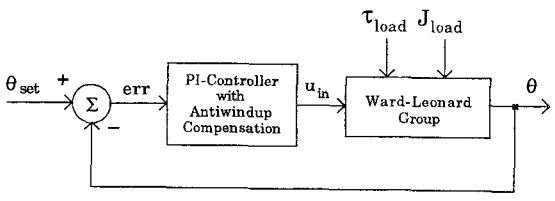

Figure 7. Position control circuit.

Beside from the drive, it contains a limiter block and the following discrete PI–controller with antiwindup compensation (Åström and Wittenmark, 1990).

```
model type PIbound

    cut in(err), out(u/-i)
    main path P < in - out >
    local x = 0.0, NextTime = 0.0, u = 0.0, ud

    parameter k = 0.06, T = 0.002, Tw = 0.01, Sample = 0.1
    parameter umax = 10.0

        when Time >= old(NextTime) then
            ud = old(x) + k * err
            u = if ud > umax then umax
                else if ud < -umax then -umax else ud
            x = old(x) + Sample * (k * err/T + Tw * (u - ud))
            NextTime = Time + Sample
        end when

end
```

Dymola treats discrete–time systems as a special case of discrete–event systems.

The overall control system model can now be assembled.

```
model WLsystem

    submodel (PIbound) PI
    submodel (WLdrive)WL

    local thset, taudi, Jldin
    output theta, omega, tau

        connect PI to WL

        thset = if Time < 50.0 then 10.0 else 5.0
        taudi = if Time < 100.0 then 0.0 else 0.01
        Jldin = if Time < 100.0 then 0.0 else 0.05

        PI.err = thset - theta
        WL.Jl = Jldin
        WL.taul = taudi
        theta = WL.theta
        omega = WL.omega
        tau = WL.tau

end
```

If expressions are used to describe the discontinuous input functions. Since the conditions of these *if* expressions are functions of *Time* alone, Dymola automatically maps these expressions into time events rather than state events.

This completes the description of the Ward–Leonard drive system. The model is quite complex. It contains both continuous and discrete states. It implicitly defines both time events and state events. Finally, it is a higher–index model. Yet, due to the object–oriented approach employed by Dymola, each component model is fairly short and easily readable, and Dymola's connection mechanisms enable the user to compose complex models in an intuitive and easily manageable fashion.

TRANSFORMATION OF EQUATIONS

Object–oriented modeling of continuous–time systems requires horizontal as well as vertical sorting of equations. The need for vertical sorting has long been recognized. This feature had been advocated in the CSSL language standard (Augustin *et al.*, 1967), and is offered by most of the contemporary continuous–system simulation languages. *Vertical sorting* means that equations can be entered in an arbitrary sequence, and the responsibility lies with the compiler to sort the equations so that no variable is being used before it has been defined. *Horizontal sorting*, on the other hand, is a child of the object–oriented world view (Elmqvist, 1978). The need for horizontal sorting of equations originates from the fact that resistors and energy transducers have multiple causalities. Whether the voltage drop across a resistor "causes" current to flow, or whether current flowing through a resistor "causes" a voltage drop, is not evident in advance. In fact, neither of the two interpretations is physically correct. Voltage drop and current flow are simply two different aspects of one and the same physical phenomenon. Causality, in this context, is a chimera, an invention of the numerical simulation algorithm.

The simulation model may require an equation of the type:

$$u = R \cdot i \tag{6a}$$

or:

$$i = \frac{u}{R} \tag{6b}$$

depending on the required numerical causality of the resistor, but the modeler should not be bothered by this problem.

It is the duty of the modeling software to support the user in formulating physical knowledge. It should protect him or her from the need to also think about the underlying numerical algorithms that are employed during simulation of the model. Horizontal sorting simply means that equations can be formulated in an arbitrary fashion. It is the responsibility of the modeling compiler to solve them symbolically in such a way that the required numerical causality results.

The world view of event descriptions has taught us to think of discrete models as sequential in nature, yet, it is beneficial to treat discrete equations in just the same way as continuous equations. In Dymola, this approach has been taken. Discrete state variables and Boolean variables can be formulated in hybrid models in exactly the same way as continuous state variables and algebraic variables. All relationships between variables are formulated in equation form that are then collected by the modeling compiler into a single monolithic block of model equations that are sorted both vertically and horizontally, i.e. discrete equations and continuous equations are sorted together. Only after this has been accomplished will Dymola look at the discrete variables once more and generate proper event descriptions for the synthesized simulation program.

CONCLUSIONS

In this paper, concepts for modeling and simulation of hybrid systems were presented. The numerical features needed by a simulation language to properly handle discontinuities are generally well understood. Exceptions are cases where the event density is very high or, in the limit, reaches infinity, such as in the descriptions of discontinuous hyperbolic partial differential equations using a method–of–lines approximation, or in the description of nonlinear control systems with an inconsistent switching characteristic. Such problems are being tackled in a current research effort (Taylor, 1993).

Event descriptions had been invented to serve the needs of numerical algorithms used for simulation of dynamical systems. While all the arguments that led to their original conceptualization are still valid, it had not been recognized before that there is no need to propagate these event descriptions all the way from the numerical simulation algorithms to the modeler's programming interface.

Event descriptions are not convenient for the modeler. They force the user to split descriptions of different aspects of one and the same physical phenomenon into different code segments in the model, and they are therefore the death blow to any attempt at object–oriented modeling. Large–scale system models that are encoded using event descriptions are difficult to develop and properly debug, and they are even harder to read and maintain.

A new methodology was presented that hides the event descriptions from the modeler, thereby offering him or her an object–oriented approach to modeling hybrid systems that is considerably more convenient to use than previously employed techniques, and that therefore can help to reduce the time needed for modeling complex hybrid systems, as well as for maintaining the models once developed.

Such descriptions are automatically translated into the format needed by the simulation environment. While it is possible to automatically translate *all* potentially discontinuous functions into proper event descriptions, this may not always be desirable. For example, the compiler will happily translate an inconsisting switching model into a seemingly correct simulation program, but the simulator will hang since it will try to handle infinitely many state events within a finite amount of simulated time, which deplorably requires an infinite amount of real time. Therefore, compiler options must be made available to give the user some control over the code that is being generated.

Dymola has been used to illustrate the underlying concepts, since semantics cannot be described without syntax. However, the message given is much more fundamental. The paper provides a description of basic mechanisms for the formulation as well as numerical treatment of hybrid models, as they arise all too often in engineering applications. The authors of this paper propose the Ward–Leonard drive example as a benchmark problem for hybrid modeling software, since the system is simple enough to be described in full, yet, exhibits most elements of a true hybrid model.

REFERENCES

Åström, K.J., and B. Wittenmark, *Computer Controlled Systems*, Second Edition, Prentice–Hall, Englewood Cliffs, N.J., 1990.

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

Brenan, K.E., S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial–Value Problems in Differential Algebraic Equations*, North–Holland, New York, 1989.

Cellier, F.E., *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*, Ph.D. Dissertation, Diss ETH No 6483, ETH Zürich, CH–8092 Zürich, Switzerland, 1979.

Cellier, F.E., "Combined Continuous/Discrete Simulation — Applications, Techniques and Tools," *Proceedings 1986 Winter Simulation Conference*, Washington, D.C., pp. 24–33, December, 1986.

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

Cellier, F.E., and H. Elmqvist, "Automated Formula Manipulation Supports Object–Oriented Continuous–System Modeling," IEEE Control System Magazine, April, 1993, in press.

Cellier, F.E., B.P. Zeigler, and A.H. Cutler, "Object–Oriented Modeling: Tools and Techniques for Capturing Properties of Physical Systems in Computer Code," *Proceedings CADCS'91 — IFAC Symposium on Computer–Aided Design in Control Systems*, Swansea, Wales, U.K., pp. 1–10, July 15–17, 1991.

DynaSim AB, *Dymola - User's Manual*, Research Park Ideon, 223 70 Lund, Sweden, 1993.

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

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

Frederick, D.K., and J.H. Taylor, *SIMNON Reference Manual*, GE Corporate Research and Development, 1989.

Gear, C.W., *Numerical Initial Value Problems in Ordinary Differential Equations*, Series in Automatic Computation, Prentice–Hall, Englewood Cliffs, N.J., 1971.

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

Mathworks, Inc., *The Student Edition of MATLAB for MS–DOS or Macintosh Computers*, Prentice–Hall, Englewood Cliffs, N.J., 1992.

Mitchell & Gauthier Associates, Inc., *Advanced Continuous Simulation Language (ACSL) — Reference Manual*, Concord, Mass, 1991.

Otter, M., "DSblock: A Neutral Description of Dynamic Systems," *OPEN–CACSD Electronic Newsletter*, **1**(3), February 28, 1992.

Pantelides, C.C., "The Consistent Initialization of Differential–Algebraic Systems," *SIAM J. Sci. Stat. Comput.*, **9**(2), pp. 213–231, 1988.

Taylor, J.H., "Toward a Modeling Language Standard for Hybrid Dynamical Systems," to appear in: *Proc. IEEE Conf. on Decision and Control*, San Antonio, Texas, Decmeber 15–17, 1993; also see Odyssey Research Associates Technical Report *A Proposed Modeling Language Standard for Hybrid Dynamical Systems*, February 1993.