The Need for Automated Formula Manipulation in Object–Oriented Continuous–System Modeling*

François E. Cellier

Department of Electr. & Comp. Engr.
University of Arizona
Tucson, AZ 85721
U.S.A.

EMail: Cellier@ECE.Arizona.Edu

Hilding Elmqvist

Simcoe
3 Grove Road
Markham, Ontario L3P 4M2
Canada

EMail: Elmqvist@Control.LTH.SE

Abstract

In this paper, automated formula manipulation is shown to be central to object–oriented continuous–system modeling. Such techniques are needed to (i) solve the causality assignment problem in modeling any kind of energy transducer, (ii) generate the equations that result from the couplings between different objects, (iii) automatically reduce structurally singular models, and (iv) take care of algebraic loops that often result from subsystem couplings, and that also occur from the reduction of structurally singular models. A new tool, Dymola 2.0, is presented that implements all of the aforementioned formula manipulation techniques, and that can be used to generate state–space models in a variety of different simulation languages (ACSL, DESIRE, and Simnon).

Introduction

The first generation of digital continuous–system simulation languages were designed to resemble analog computer "programs." They were block–diagram languages with adders, integrators, multipliers, and potentiometers used as their basic building blocks. This was done in order to "ease" the transition from analog to digital simulation technology. It took the modelers of that era several years to realize that programming an analog computer hadn't been that convenient after all and that, by making digital simulation languages resemble analog programs, they actually made their task unnecessarily hard. Analog computer programming had been dictated by the technology in use, it wasn't designed to suit the human programmer.

Digital technology is not bound by the same limitations as analog technology. There is considerably more flexibility in designing digital programs. The next generation of simulation languages started out from the mathematics of numerically solving sets of ordinary differential equations. It turns out that most numerical integration algorithms are designed to solve so–called state–space models of the type

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{1}$$

Continuous–system simulation languages used today have been designed to facilitate the formulation of state–space models. It was quickly recognized that the same expressions may reappear in several state equations, and that it is more efficient from a computational point of view (and also more convenient) to assign these expressions to auxiliary (algebraic) variables. Consequently, the extended state–space model used in simulation languages takes the form

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

with the additional restriction that the auxiliary variables, $\mathbf{z}$, must not depend algebraically upon each other in a mutual way, i.e., that no *algebraic loops* are contained in the model. As an additional bonus, simulation language designers added an *equation sorter* that enables the user to specify the model equations in an arbitrary sequence and that thereby also supports the use of macros. Macros are used to describe subsystems in a compact fashion. They are invoked like subroutines, but their treatment within the simulation language is very different from that of a subroutine. The simulation compiler inserts the statements that are formulated within a macro into the simulation program at the place of its call. This happens *before* the equation sorter is activated. This is important since, once an executable statement sequence has been established, the statements that were extracted from different macros are now mixed[4].

It is important to realize that also simulation languages of the CSSL–type[2] that are in use today are technology–based. This time, it is not the technology of electronic and/or mechanical components that dictates the modeling methodology, instead, today's simulation languages are designed to suit the mathematical technology of numerical integration algorithms. This fact is illustrated in the following example. Figure 1 shows a simple passive electrical circuit:

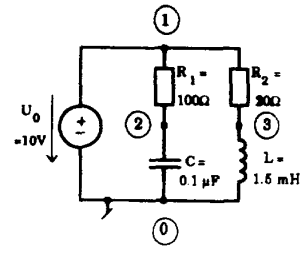

Figure 1. Simple passive electrical circuit.

In a CSSL-type simulation language, this circuit could be represented as:

$$\begin{aligned} U0 &= f(t) \\ uC &= \mathrm{INTEG}(iC/C, uC0) \\ iL &= \mathrm{INTEG}(uL/L, iL0) \\ uR2 &= R2 * iL \\ uR1 &= U0 - uC \\ iC &= uR1/R1 \\ uL &= U0 - uR2 \\ i0 &= iC + iL \end{aligned}$$

* Research partially supported by NASA–Ames / University of Arizona Cooperative Agreement No. NCC 2–525, "A Simulation Environment for Laboratory Management by Robot Organizations".

which corresponds to the block diagram of Figure 2:

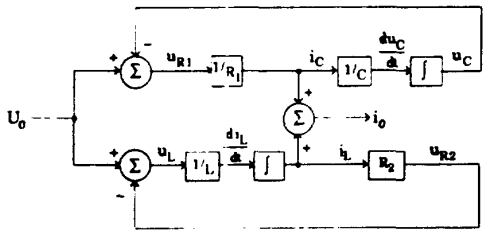

Figure 2. Block diagram of electrical circuit.

The block diagram shows the *computational causality* of the model. The computational causality of a model determines how the physical laws that are encoded in the model equations must be interpreted in order to obtain a program that can be executed on a sequential machine using existing numerical algorithms. In the above example, Ohm's law is utilized differently when applied to the two resistors, R_1 and R_2. In the case of R_1, the current through the resistor, i_{R_1} is computed from the voltage across the resistor, u_{R_1}, which is computed elsewhere, whereas in the case of R_2, the reverse is true. Obviously, both equations describe one and the same physical phenomenon.

No modeler would normally fall upon the idea to represent this circuit by the set of equations:

$$
\begin{aligned}
U0 &= f(t) \\
iC &= C * \mathrm{DERIV}(uC) \\
uL &= L * \mathrm{DERIV}(iL) \\
uR1 &= R1 * iC \\
uC &= U0 - uR1 \\
uR2 &= U0 - uL \\
iL &= uR2/R2 \\
i0 &= iC + iL
\end{aligned}
$$

which would correspond to the block diagram of Figure 3:

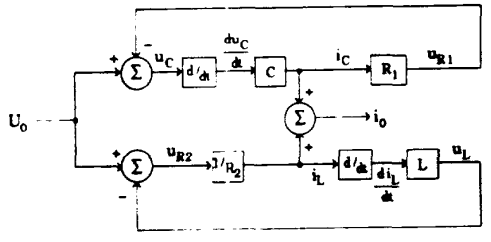

Figure 3. Alternative block diagram of electrical circuit.

although both descriptions are completely equivalent from an analytic point of view. The fact is that modelers have learned to avoid the DERIV operator at all cost, since it is *numerically* easier to integrate variables than to differentiate them (at least as long as explicit numerical algorithms are used, which is always the case in today's simulation software).

This example demonstrates the intimate interrelation of the modeling methodology supported by today's simulation software and the characteristics of the underlying numerical algorithms. From the point of view of user convenience, there is no difference between the two formulations. It is not suggested here that it would be in any way more advantageous to formulate models in *differential causality* (i.e., by use of the DERIV operator) rather than in *integral causality* (i.e., by use of the INTEG operator). Yet, it is demonstrated below that the familiar state–space model is not the most convenient way to specify a model.

The above equations, or rather assignment statements, are not obvious when inspecting the circuit shown on Figure 1. The reason is that basic electrical laws have been transformed into unfamiliar forms. The familiar forms are given below as true equations:

$U_0 = f(t)$	{Voltage source.}
$C \cdot \frac{d(u_C)}{dt} = i_C$	{Capacitor. Law of capacitance.}
$L \cdot \frac{d(i_L)}{dt} = u_L$	{Inductor. Law of inductance. }
$u_{R_1} = R_1 \cdot i_C$	{Resistor. Ohm's law.}
$u_{R_2} = R_2 \cdot i_L$	{Resistor. Ohm's law.}
$U_0 = u_{R_1} + u_C$	{Kirchoff's voltage law.}
$U_0 = u_{R_2} + u_L$	{Kirchoff's voltage law.}
$i_0 = i_C + i_L$	{Kirchoff's current law.}

There is a much closer correspondence between this formulation and the circuit diagram. These equations can be written down directly by inspecting the circuit diagram. The correctness of the model equations is thus promoted. In this paper, a modeling tool is introduced that allows the user to formulate his or her model in such an equation–based form.

The Causality Assignment Problem

The above example contains two *objects* of the *class* resistor. Yet, in the familiar CSSL–type formulation, the equations used to describe these two objects are different. In the case of resistor R_2, the current flowing through the resistor seems to "cause" a potential drop across the resistor. In the case of R_1, the potential drop across the resistor seems to "cause" current to flow through the resistor. Moreover, the causalities for the two resistors change if the model as a whole is formulated in differential rather than integral causality. Quite obviously, *computational causality* is not a physical phenomenon at all, but is simply yet another artifact of the underlying numerical algorithm.

It is rather inconvenient that the user must determine the (numerically) correct causality of the dissipative elements, or more generally, the causality of all energy transducers (transformers exhibit exactly the same problem as resistors). It would be much nicer if objects, such as a resistor, could be described once and for all in terms of their *physical* properties and their interactions with the environment. In case of the resistor, such an approach would call for a description of the resistor *itself* (Ohm's law) and a description of how this equation interacts with other equations of the neighboring components.

However, object–oriented continuous–system modeling[7] is much more than just a matter of convenience. State–space models suggest that each state variable changes with time according to some law that is expressed in the corresponding state equation. But why does this happen? The voltage across a capacitor doesn't change with time unless it has a good reason for doing so. Physics is a matter of *trade*. The only tradable goods are mass and energy. Consequently, it would be much safer if the modeling environment were to enable the user to formulate mass balances and energy balances rather than state equations. If a state equation is formulated incorrectly, a CSSL–type simulation language[2] will happily accept the incorrect equation, and trade it for beautiful multi–colored graphs that may even look plausible[5].

The modeling language Dymola[9] incorporates these concepts. In Dymola, a resistor can be described as follows:

```
model type resistor
    cut WireA(Va/i), WireB(Vb/-i)
    main path P < WireA - WireB >
    local u
    parameter R = 1.0
        u = Va - Vb
        u = R * i
end
```

Ohm's law is described in the usual way. It involves the parameter R, which has a default value of 1.0, the local variable u, and the terminal variable i. The *cut* and *path* declarations are used to describe the interface to the outside world. Additional equations are formulated to specify the relations between the local variables and the terminal variables.

Of course, the chosen approach also calls for a general mechanism to describe the couplings between different interconnected objects. In Dymola, the above example circuit could be represented as follows:

```
model circuit

    submodel (vsource) U0
    submodel (resistor) R1(R = 100.0), R2(R = 20.0)
    submodel (capacitor) C(C = 0.1E-6)
    submodel (inductor) L(L = 1.5E-3)
    submodel Common
    node n0, n1, n2, n3
    input u
    output y1, y2

    connect Common at    n0,
            U0      from n0 to n1,
            R1      from n1 to n2,
            C       from n2 to n0,
            R2      from n1 to n3,
            L       from n3 to n0

    U0.V = u
    y1 = C.u
    y2 = L.i

end
```

The *submodel* declaration instantiates objects from classes. For example, two objects of type *resistor* are instantiated, one named $R1$ with a parameter value of $R = 100.0\ \Omega$ and the other named $R2$ with a parameter value of $R = 20.0\ \Omega$. The *connect* statement is used to describe the interconnection between objects. Notice that the connecting equations (Kirchhoff's laws) are not explicitly formulated at all. They are automatically generated at compile time from the topological description of the interconnections.

Upon entering the model, Dymola immediately instantiates all submodels (objects) from the model types (classes). It then extracts the formulated equations from these objects, and expands them with the coupling equations that are being generated from the description of the interconnections between objects. For the above example, the result of this operation is the following:

U0	$V = Vb - Va$
R1	$u = Va - Vb$
	$u = R * i$
R2	$u = Va - Vb$
	$u = R * i$
C	$u = Va - Vb$
	$C * \mathrm{der}(u) = i$
L	$u = Va - Vb$
	$L * \mathrm{der}(i) = u$
Common	$V = 0$
circuit	$U0.V = u$
	$y1 = C.u$
	$y2 = L.i$
	$R1.Vb = C.Va$
	$C.i = R1.i$
	$R1.Va = R2.Va$
	$U0.Vb = R1.Va$
	$R2.i + R1.i = U0.i$
	$R2.Vb = L.Va$
	$L.i = R2.i$
	$C.Vb = L.Vb$
	$U0.Va = C.Vb$
	$Common.V = U0.Va$

The first 10 of these equations are extracted from the submodels. The next three equations are extracted from the circuit model. The last 10 equations represent Kirchhoff's laws. These equations are automatically being generated from the connect statements that describe the interconnections between the objects.

The structure of the equations needs to be examined in order to determine which variable to solve for in each equation. In addition, the equations need to be sorted into a correct computational order. If this is not possible due to mutual dependencies, minimal systems of equations, that need to be solved simultaneously, should be isolated. These problems are naturally solved by use of graph-theoretical algorithms[9]. The structure of equations and variables is represented by a *bipartite graph*. The problem of associating each equation with one variable is called the *assignment* problem. Algorithms with execution time depending linearly on the number of nodes can be found in Wiberg[17]. The sorting problem is referred to as finding the *strong components* of the graph. Tarjan[16] has designed an algorithm with linear time dependency based on a depth-first traversal of the graph.

The *partition eliminate* command in Dymola utilizes these algorithms to solve the causality assignment problem. It also eliminates trivial equations of the type

$$a = b \tag{3}$$

The result of this operation is as follows:

Common	$[L.Vb] = 0$
U0	$circuit.u = [R2.Va] - L.Vb$
C	$u = [Va] - L.Vb$
R1	$[u] = R2.Va - C.Va$
	$u = R * [i]$
C	$C * [\mathrm{der}(u)] = R1.i$
R2	$[u] = R * L.i$
	$u = Va - [L.Va]$
L	$[u] = Va - Vb$
	$L * [\mathrm{der}(i)] = u$
circuit	$L.i + R1.i = [U0.i]$
	$[y1] = C.u$
	$[y2] = L.i$

In each equation, the variable to be solved for is marked by square brackets. Notice the different causalities for the two resistors.

At this point, further formula manipulation can be used to solve the equations in order to generate a state-space model. The algorithm used for solving equations symbolically works on an internal representation of equations, called a syntax tree. In order to solve equations, Dymola recursively applies certain transformation and simplification rules to the tree representation.

Dymola has rules about the inverse of certain functions and handles the case of several linear occurrences of the unknown variable. Solving the following equation for x:

$$\exp(a + \sin([x]/b + c * [x] - d) * (\exp(e) + 1)) * *2 - f = 2 * g \quad (4)$$

gives the result:

$$x = (\arcsin((\ln(\mathrm{sqrt}(2 * g + f)) - a)/(\exp(e) + 1)) + d)/(1/b + c) \quad (5)$$

More about symbolic formula manipulation can, for example, be found in Davenport, Siret, and Tournier[8].

For the above circuit example, the result of the command:

```
> output solved equations
```

is as follows:

Common	$L.Vb = 0$
U0	$R2.Va = circuit.u + L.Vb$
C	$Va = u + L.Vb$
R1	$u = R2.Va - C.Va$
	$i = u/R$
C	$\mathrm{der}(u) = R1.i/C$
R2	$u = R * L.i$
	$L.Va = Va - u$
L	$u = Va - Vb$
	$\mathrm{der}(i) = u/L$
circuit	$U0.i = L.i + R1.i$
	$y1 = C.u$
	$y2 = L.i$

Finally, the state–space model can be automatically encoded as a text file in any one of a list of simulation languages. Notice that Dymola is not a simulation program in its own right. It does not provide for any simulation support at all. Dymola can be viewed as a sophisticated *macro processor* since it can be used as a frontend to a simulation language and thereby (among other things) assumes the role of its macro processor. Dymola can also be viewed as a *model generator* since it can generate models for a variety of different simulation languages. The currently supported languages are ACSL[18], DESIRE[11], and Simnon[16]. However, the most adequate interpretation is to view Dymola as a *modeling language*. Dymola has been designed to facilitate the object–oriented formulation of models of complex continuous systems. The user interface (language definition) of Dymola is much less technology–driven than CSSL–type simulation languages. It is designed to increase user–convenience. The Dymola software, on the other hand, is strongly technology–driven since it generates a state–space model whenever possible. This is a deliberate choice. It would be possible to make the Dymola program convert a Dymola model into a description that could then be simulated by use of a differential/algebraic equation (DAE) solver[3]. The decision to generate a state–space model was based on efficiency considerations. It is usually more efficient to manipulate the model at compile time to generate code that executes fast than to lay the burden of model manipulation on the numerical algorithm of the run–time program (a DAE solver).

The Algebraic Loop Problem

It was mentioned earlier that simulation languages do not permit mutual algebraic relations between variables. This is due to the fact that, in such a case, the equation sorter cannot determine a proper execution sequence of the model statements. With the two equations:

$$y = f(x) \quad (6a)$$
$$x = g(y) \quad (6b)$$

x must be known before y can be computed from the first equation, but y must be known in order to compute x from the second equation. Consequently, neither of the two equations can be computed without the other.

Algebraic loops among variables *within* a model sometimes mean bad modeling, or rather, a bad choice of variables. However, algebraic loops that are the result of interconnections between different objects occur frequently and are unavoidable. A simple example of this type is the voltage divider shown on Figure 4:

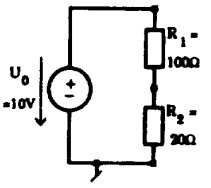

Figure 4. Voltage divider.

The voltage divider can by coded in Dymola as follows:

```
model divider

    submodel (vsource) U0
    submodel (resistor) R1(R = 100.0), R2(R = 20.0)
    submodel Common
    input u
    output y

    connect Common – U0 – R1 – R2 – Common

    U0.V = u
    y = R2.u

end
```

When this model is entered into Dymola, the following set of equations is generated:

U0	$V = Vb - Va$
R1	$u = Va - Vb$
	$u = R * i$
R2	$u = Va - Vb$
	$u = R * i$
Common	$V = 0.0$
divider	$U0.V = u$
	$y = R2.u$
	$R1.Va = U0.Vb$
	$R1.i = U0.i$
	$R2.Va = R1.Vb$
	$R2.i = R1.i$
	$Common.V = R2.Vb$
	$U0.Va = Common.V$

After the *partition eliminate* operation has been issued, the solved equations can be displayed. The result of that operation is as follows:

Common	$R2.Vb = 0$
U0	$Vb = divider.u + R2.Vb$

System of 4 equations. Unknown variables:
$R1.Vb$
$divider.y$
$U0.i$
$R1.u$

-R2	$divider.y = [R1.Vb] - Vb$
-	$[divider.y] = R \bullet U0.i$
-R1	$u = R \bullet [U0.i]$
-	$[u] = U0.Vb - Vb$

End of system of equations.

Solved system of equations:
$R1.Vb = (R1.R \bullet R2.Vb + R2.R \bullet U0.Vb)/(R1.R + R2.R)$
$divider.y = (R2.R \bullet U0.Vb - R2.R \bullet R2.Vb)/(R1.R + R2.R)$
$U0.i = (U0.Vb - R2.Vb)/(R1.R + R2.R)$
$R1.u = (R1.R \bullet U0.Vb - R1.R \bullet R2.Vb)/(R1.R + R2.R)$
End solved system of equations.

The simple fact that this circuit contains two series–connected resistors results in a system of simultaneous equations (an algebraic loop) involving four variables and four equations. The causality assignment problem can no longer be solved in a unique fashion, which is always an indication of algebraic loops. Dymola detects the algebraic loop, isolates the involved equations, determines the involved variables, discovers that the algebraic loop is linear, and therefore is able to solve it at once by symbolic formula manipulation. Further simplifications are possible. Dymola can be set up to find common sub–expressions and introduce auxiliary variables for them. This reduces the amount of computations needed. Equations of the type:

$$a = 0 \tag{7}$$

can be eliminated from the model, and in all other equations, terms multiplied by a can also be eliminated. Finally, equations that evaluate a variable, which is neither used in any other equation nor declared as an output variable, are surplus equations that can be omitted from the model.

With these two additional simplifications, the above model is reduced to a single equation:

$$divider.y = R2.R \bullet divider.u/(R1.R + R2.R)$$

which is the well–known voltage divider equation.

Obviously, not all algebraic loops are linear. Nonlinear algebraic loops cannot generally be solved by formula manipulation. Also, it can happen that a single linear algebraic loop contains many equations and many variables, in which case the solved set of equations may look formidable. In such cases, it may still be necessary to employ a numerical, iterative method, such as a Newton–Raphson type algorithm for a subsystem of equations.

Structurally Singular Models

Structurally singular problems are systems that contain more energy storing elements than eigen modi. A structurally singular linear electrical circuit contains more capacitors and/or inductors than indicated by the order of its transfer function. Structural singularities are related to index $n, n > 1$ DAEs[8,12].

Structural singularities can easily be detected as a byproduct of the algorithm that determines the computational causality. If, during causality assignment, any of the integrators (energy storage elements) assumes differential rather than integral causality, the model is structurally singular.

As in the case of linear algebraic loops, structural singularities within models often indicate bad modeling, or rather a poor selection of variables. However, structural singularities that are caused by interconnections between objects are frequent and unavoidable. This fact can be demonstrated by the simple circuit shown on Figure 5.

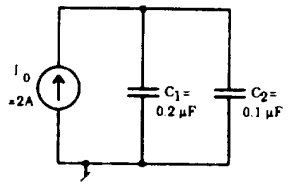

Figure 5. Parallel capacitor circuit.

This circuit can be modelled in Dymola as follows:

```
model parcap

    submodel (capacitor) C1(C = 0.2E-6), C2(C = 0.1E-6)
    submodel (csource) I0
    submodel Common
    input i
    output y

    connect Common - I0 - (C1//C2) - Common

    I0.II = i
    y = C1.u

end
```

When this model is entered into Dymola, the following equations are generated:

C1	$u = Va - Vb$
	$C*\mathrm{der}(u) = i$
C2	$u = Va - Vb$
	$C*\mathrm{der}(u) = i$
I0	$V = Vb - Va$
	$i = II$
Common	$V = 0$
parcap	$I0.II = i$
	$C2.Vb = C1.Vb$
	$Common.V = C2.Vb$
	$I0.Va = Common.V$
	$C1.Va = I0.Vb$
	$C2.Va = C1.Va$
	$C1.i + C2.i = I0.i$

Partitioning this problem leads to the following warning:

Singular problem.

Unassigned variables:
$C2.i$

Redundant equations:

parcap	$C2.Va = C1.Va$

and the generated equations are:

C1	$u = [Va] - Vb$
	$C \bullet [\mathrm{der}(u)] = i$
C2	$u = [Va] - Vb$
	$C \bullet [\mathrm{der}(u)] = i$
I0	$[V] = Vb - Va$
	$[i] = II$
Common	$[V] = 0$
parcap	$[I0.II] = i$
	$C2.Vb = [C1.Vb]$
	$Common.V = [C2.Vb]$
	$[I0.Va] = Common.V$
	$C1.Va = [I0.Vb]$
	$C2.Va = C1.Va$
	$[C1.i] + C2.i = I0.i$

Dymola assumes by default that the state variables of the model are all variables that appear differentiated. Due to the fact that the target simulation language is expected to make use of an explicit integration technique, all state variables can automatically be declared as *known variables* according to equation (1).

It is possible to get around the singularity by telling Dymola explicitly that one of the two so–called state variables that were introduced by default is, in fact, not a state variable at all. This can be accomplished by declaring:

```
> variables unknown C2.u
> variables known C2.deru
```

Now, the equations can be repartitioned, and after eliminating the trivial assignments, and after sorting and solving them, the following set of equations is obtained:

Common	$C1.Vb = 0$
C1	$I0.Vb = u + Vb$
C2	$u = I0.Vb - C1.Vb$
	$i = C*\mathrm{der}(u)$
parcap	$C1.i = i - C2.i$
C1	$\mathrm{der}(u) = i/C$
I0	$V = Vb - C1.Vb$
parcap	$y = C1.u$

It can be clearly seen that one of the two differential equations now assumes differential causality rather than integral causality.

While this is a possible solution to the dilemma, it is not a very good one, since it forces the subsequent simulation to numerically differentiate the variable $C2.u$ in order to compute $C2.i$, which is unnecessary. There exists a (linear) algebraic relationship between the two so–called state variables, i.e., the two outputs of the integrators. More precisely:

$$C2.u = C1.u \tag{8a}$$

By differentiating equation (8a), the following equation is obtained:

$$\mathrm{der}(C2.u) = \mathrm{der}(C1.u) \tag{8b}$$

One method is to replace equation (8a) by equation (8b), and thereby remove the structural singularity. The constraint is thus removed, and the voltages of the capacitors are integrated separately. It is then important to assign initial values that are consistent with the removed constraint. This approach has the drawback that numerical inaccuracy might introduce drift in such a way that the removed constraint is no longer valid after the simulation has proceeded for a while[13].

The approach taken in Dymola is to retain all constraints. The dimension of the state vector is reduced. Instead, the removed state variables are solved from the constraints. The derivatives of the removed state variables also need to be computed. Equations for those are added by differentiating the constraints.

Pantelides[14] has designed an algorithm for determining which equations need to be differentiated. It is a graph–theoretical algorithm that uses the dependency structure of the equations. This algorithm has been implemented in Dymola. When the *differentiate* command is entered, Dymola uses the algorithm to augment the set of equations with symbolically differentiated versions of some of the equations. The algorithm assumes that all state variables are *known*. It then looks for constraints between these variables. Note, that there might be a chain of equations with auxiliary variables involved. All equations in such a dependency chain must be differentiated.

This process is repeated because there might be second order derivatives implying that differentiated variables are considered known. The added differentiated equations might introduce constraints on these differentiated variables, which means that these equations have to be differentiated once more.

Once the *differentiate* command has been issued, Dymola no longer assigns any variables to the set of state variables automatically, but leaves it up to the user to declare, which variables are to be used as state variables.

The parallel capacitor problem can be tackled using the following set of commands:

```
> differentiate
> variables known C1.u
> partition
> output equations
```

which leads to the following set of equations:

C1	$u = [Va] - Vb$
	$C * [deru] = i$
C2	$[u] = Va - Vb$
	$C * deru = [i]$
I0	$[V] = Vb - Va$
	$[i] = II$
Common	$[V] = 0$
parcap	$[I0.II] = i$
	$C2.Vb = [C1.Vb]$
	$Common.V = [C2.Vb]$
	$[I0.Va] = Common.V$
	$C1.Va = [I0.Vb]$
	$[C2.Va] = C1.Va$
	$[C1.i] + C2.i = I0.i$
C1	$deru = [derVa] - derVb$
parcap	$C2.derVb = [C1.derVb]$
Common	$[derV] = 0$
C2	$[deru] = derVa - derVb$
parcap	$Common.derV = [C2.derVb]$
	$[C2.derVa] = C1.derVa$

The last six equations of the above set are those that have been added by applying the Pantelides algorithm to this model.

By declaring $C1.u$ as a known variable, the causality assignment algorithm inside Dymola knows that it doesn't need to find an equation to evaluate $C1.u$, and the model generator inside Dymola knows that it needs to generate a state equation for this variable. For example in the case of ACSL, a statement of the type:

$$C1\_u = \mathrm{INTEG}(C1\_deru, 0.0)$$

will be added to the set of generated equations.

The commands:

```
> partition eliminate
> output solved equations
```

will lead to the following set of equations:

Common	$derV = 0$
parcap	$C2.derVb = Common.derV$
	$C1.derVb = C2.derVb$
	$I0.II = i$

System of 6 equations. Unknown variables:
$C1.i$
$C2.i$
$C2.deru$
$C2.derVa$
$C1.derVa$
$C1.deru$

–	$[C1.i] + C2.i = I0.II$
–C2	$C * deru = [i]$
–	$[deru] = derVa - derVb$
–parcap	$[C2.derVa] = C1.derVa$
–C1	$deru = [derVa] - derVb$
–	$C * [deru] = i$

End of system of equations.

Solved system of equations:

$Det1 = C1.C + C2.C$

$C1.i = (C1.C * I0.II + C2.C * C1.C * C2.derVb$
$\quad -C2.C * C1.C * C1.derVb)/Det1$

$C2.i = (C2.C * I0.II - C2.C * C1.C * C2.derVb$
$\quad +C2.C * C1.C * C1.derVb)/Det1$

$C2.deru = (I0.II - C1.C * C2.derVb + C1.C * C1.derVb)/Det1$

$C2.derVa = (I0.II + C2.C * C2.derVb + C1.C * C1.derVb)/Det1$

$C1.derVa = (I0.II + C2.C * C2.derVb + C1.C * C1.derVb)/Det1$

$C1.deru = (I0.II + C2.C * C2.derVb - C2.C * C1.derVb)/Det1$

End solved system of equations.

Common	$C1.Vb = 0$
C1	$I0.Vb = u + Vb$
I0	$V = Vb - C1.Vb$
C2	$u = I0.Vb - C1.Vb$

which can be used to automatically generate a simulation program for either ACSL, DESIRE, or Simnon.

Applications

This section describes some typical modeling situations where symbolic model manipulation is needed.

When modeling a *mechanical system*, the technique of *free body diagrams* is utilised. The introduced forces and torques are terminal variables that are structured into cuts to facilitate the description of the mechanical topology. Connecting mechanical links and joints introduces constraints on positions and velocities, which implies that the degrees of freedom of the interconnected system are reduced, i.e., the dimension of the state vector of the interconnected system is smaller than the sum of the dimensions of the state vectors of the subsystems.

A simple example is the model of a body in two dimensions for which one end point is attached to a fixed rotational joint. The unconstrained body has three degrees of freedom. It can translate in x and y directions, and it can rotate around its z–axis. Thus, a state–space model of an unconstrained body must contain six first–order ODEs. Due to the connection with the rotational joint, the lever is restricted in its freedom to move. It can no longer translate at all. It can only rotate around the joint. Consequently, a state–space model of the constrained body must contain two first–order ODEs. The degrees of freedom are reduced from three to one.

The model type that describes the body irrespective of the environment it operates in must describe the unconstrained body. Consequently, it may contain either two instances of Newton's translational law and one instance of Newton's rotational law, equivalent descriptions using the d'Alembert principle, a direct formulation of the energy balance equations, or finally, a description of power flow through the system (e.g. using a bond graph notation[4,6]). In either formulation, an instantiation of the unconstrained body will result in a sixth–order state–space model.

The model type that describes the joint doesn't contain any dynamics at all, since the joint by itself doesn't move around. When the unconstrained body is connected through the joint to the wall, four constraints (two explicit positional and two deduced velocity constraints) are introduced. The resulting model is thus structurally singular. By applying the Pantelides algorithm (differentiation), it is possible to get rid of the structural singularity. In the process, the number of state equations is reduced from six to two. By choosing the angular position, θ, and the angular velocity, ω, as the two remaining state variables, a system of equations arises, i.e., the resulting model contains a linear algebraic loop that can be solved by formula manipulation. The solution is of the form:

$$\mathrm{der}(\omega) = \ldots/(J + m \cdot d \cdot d) \tag{9}$$

where J is the inertia of the body relative to its point of gravity, d is the distance from the center of gravity to the joint, and m is the mass of the body. The formula for how the inertia changes due to translation ($J + m \cdot d \cdot d$) is thus automatically obtained.

Thermodynamic systems and *chemical reaction dynamics* are modeled by defining *control volumes* and introducing terminal variables in cuts corresponding to e.g. the pipes between different components. The topology is typically described as separate subgraphs by following the different flows in the system (steam, water, etc).

As an example, consider a superheater in a thermal power plant. A model for the steam is:

$$\mathrm{der}(E) = Q_{in} - W \cdot (h - h_{in}) \tag{10a}$$

$$E = V \cdot r \cdot h \tag{10b}$$

$$r = g(h) \tag{10c}$$

where E denotes the stored energy, Q_{in} describes the incoming heat flow, W is the mass flow rate of steam, h is the enthalpy of steam in the superheater, h_{in} describes the enthalpy of incoming steam, V denotes the volume, and r is the density. The function g describes steam properties and is typically implemented as a table look–up function.

If E is chosen as the state variable (default selection), a nonlinear system involving equations (10b) and (10c) has to be solved for h and r. An alternative approach is to select the enthalpy h as the state variable. The differentiation algorithm in Dymola determines that the equations (10b) and (10c) have to be differentiated. A two–dimensional linear system of equations results. Its solution produces:

$$\mathrm{der}(h) = derE/(\mathrm{gDER}(h) \cdot V \cdot h + V \cdot r) \tag{11}$$

where $\mathrm{der}(h)$ is a true state derivative, whereas $derE$ is an algebraic variable with a special name[12]. The existence of a function gDER is assumed that returns the partial derivative of the function g with respect to its argument.

It is not obvious which state variable selection is preferable. If the function gDER exists, the selection of h as a state variable probably gives more efficient computations. If gDER is not available, the former approach may be more appropriate. The point is that the modeler doesn't need to manually perform the required formula manipulations depending on which state variable is selected. The model contains only the fundamental physical equations. This makes modeling a considerably safer enterprise.

A similar situation occurs when modeling *active electronic circuits*[4]. A bipolar transistor model contains three junction diode models. Each of those models contains a nonlinear capacitor. Simplified model equations are:

$$\mathrm{der}(q_c) = i_c \tag{12a}$$

$$q_c = k_1 \cdot u_d^{k_2} + k_3 \cdot i_d + k_4 \tag{12b}$$

$$i_d = k_5 \cdot \exp(u_d) + k_6 \cdot u_d + k_7 \tag{12c}$$

where q_c is the charge, i_c is the capacitive current, i_d is the diode current, u_d is the voltage, and $k_1, \ldots, k_7$ are parameters. A choice of q_c as the state variable leads to a *nonlinear* system of equations in the variables u_d and i_d that must be solved iteratively. In an alternative approach, u_d can be chosen as the state variable. In this case, the Pantelides algorithm must be applied. After differentiation, a *linear* system of equations in the variables $\frac{du_d}{dt}$ and $\frac{di_d}{dt}$ results that can be solved by formula manipulation.

The true bipolar transistor equations are, in fact, much more complicated than indicated above. It is thus a relief for the modeler not to have to perform the differentiations by hand, and automated differentiation certainly promotes model correctness.

Summary and Conclusions

In this paper, it was demonstrated how sophisticated automated formula manipulation can be used to automatically generate state-space models from an object-oriented description of a physical system. It was shown that the two major complications, algebraic loops and structural singularities, occur frequently as a consequence of couplings between submodels (objects), and that these difficulties can often be dealt with by automated formula manipulation. All structural singularities can be reduced to systems of simultaneous algebraic equations[12], and small linear systems of equations can be solved explicitly.

The examples chosen in this paper were all very simple, and were mostly selected from the class of passive linear electrical circuits. However, the advocated techniques have been successfully applied to considerably more complex systems, and to systems stemming from various application areas, such as mechanics, thermodynamics, and chemical reaction kinetics. Many sophisticated examples can be found in the literature[1,4,7,9,15]. The selection of examples used in this paper was dictated partly by space considerations and partly by the desire to isolate the individual types of advocated formula manipulation techniques.

A software tool, Dymola, was presented in which the various formula manipulation techniques have been implemented. Dymola is an object-oriented continuous-system modeling language and a model manipulator that can be used to generate models in several simulation languages.

References

[1] M. Amrhein (1990), "Modeling of Chemical Reaction Networks Using Bond Graphs," Thesis, Dept. of Electr. & Comp. Engr., University of Arizona, Tucson, Ariz.

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

[3] K. E. Brenan, S. L. Campbell, and L. R. Petzold (1989), *Numerical Solution of Initial-Value Problems in Differential Algebraic Equations*, North-Holland, New York.

[4] F. E. Cellier (1991), *Continuous System Modeling*, Springer Verlag, New York.

[5] F. E. Cellier (1992), "Bond Graphs — The Right Choice for Educating Students in Modeling Continuous-Time Systems," *Proceedings 1992 Western Simulation MultiConference on Simulation in Engineering Education*, January 22–24, Newport Beach, California, pp. 123–127.

[6] F. E. Cellier (1992), "Hierarchical Nonlinear Bond Graphs: A Unified Methodology for Modeling Complex Physical Systems," *Simulation*, March issue, in press.

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

[8] J. H. Davenport, Y. Siret, and E. Tournier (1988), *Computer Algebra: Systems and Algorithms for Algebraic Computation*, Academic Press, New York.

[9] H. Elmqvist (1978), *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.

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

[11] G. A. Korn (1989), *Interactive Dynamic-System Simulation*, McGraw-Hill, New York.

[12] S. E. Mattsson and G. Söderlind (1992), "A New Technique for Solving High-Index Differential-Algebraic Equations," *Proceedings CACSD'92*, March 17-19, Napa, California.

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

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

[15] U. Piram (1991), "Numerical Investigation of Second Sound in Liquid Helium," Thesis, Dept. of Electr. & Comp. Engr., University of Arizona, Tucson, Ariz.

[16] R. E. Tarjan (1972), "Depth First Search and Linear Graph Algorithms," *SIAM J. Comp.*, 1, pp. 146–160.

[17] T. Wiberg (1977), *Permutation of an Unsymmetric Matrix to Block-Triangular Form*, Ph.D. dissertation, Dept. of Information Processing, University of Umea, Umea, Sweden.