"Relaxing" — A Symbolic Sparse Matrix Method Exploiting the Model Structure in Generating Efficient Simulation Code

Martin Otter
Inst. für Robotik & Systemdynamik
DLR Oberpfaffenhofen
D–82230 Wessling
Germany
Martin.Otter@DLR.DE

Hilding Elmqvist
Dynasim AB
Research Park Ideon
S–223 70 Lund
Sweden
Elmqvist@Dynasim.SE

François E. Cellier
Dept. of Electr. & Comp. Engr.
University of Arizona
Tucson, AZ 85721
U.S.A.
Cellier@ECE.Arizona.Edu

ABSTRACT

This paper presents a new method for symbolically solving large sets of algebraically coupled equations as they are frequently encountered in the formulation of mathematical models of physical systems in object–oriented modeling. The method, called "relaxing," enables the modeler to exploit the special matrix structure of the type of system under study by simply placing the keyword **relax** at appropriate places in the model class libraries. This procedure defines an evaluation sequence for a sparse matrix Gaussian elimination scheme. The method is demonstrated at hand of several broad classes of physical systems: drive trains, electrical circuits, and tree–structured multibody systems. In particular, relaxing allows a model compiler, such as Dymola, to start from a declarative, object–oriented description of the model, and to automatically derive the recursive $O(f)$ algorithm used in modern multibody programs.

Keywords: Sparse matrices; symbolic formulae manipulation; object–oriented modeling; relaxing; tearing.

1. INTRODUCTION

When deriving mathematical models of complex physical systems, it is essential that the modeler has the option to describe submodels of subsystems independently, and connect them topologically in the same fashion as he or she would connect real subsystems in a laboratory. For example, when modeling a large electronic circuit consisting of many circuit elements, it is extremely inconvenient for the modeler to derive a state–space model manually from the circuit diagram. Instead, he or she should have the possibility to invoke component models from a component model library, and, for components not available in any library yet, describe the laws that govern each component class one at a time, and then connect these component models in the same way as the real components are connected in the circuit. Block diagram editors, such as SIMULINK, which force the modeler to first reformulate the problem at hand into a connection of input/output blocks, are not suitable for such purposes, except to describe the higher echelons of a control system architecture [6, 21].

In an object–oriented modeling system, such as Dymola [8, 9] or Omola [1, 2], models are described in a declarative way, i.e., only the local properties of objects and the connections among the objects are defined. The connection structure implicitly defines the relationship between the variables of the submodels, and leads to additional equations. Collecting all the equations together usually results in a large set of differential and algebraic equations. Efficient graph–theoretical algorithms are available to sort the equations and variables into a form that contains algebraic loops of minimal dimensions [7, 8, 17].

For models of physical systems, such as electrical circuits, mechanical, or hydraulic systems, algebraic loops of minimal dimensions are often still quite large. On the other hand, derivation from an object–oriented model description naturally leads to systems of equations that are usually very sparse, i.e., only few of the variables are present in any one equation. The **A** matrix of a linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

may contain only 5–10% non–zero elements, or even less. In order to solve such a system of equations in an efficient way, the used algorithm has to take into account the sparsity of the system when applying Gaussian elimination. If nothing more is known about such a system, this task is quite hard: on the one hand, the solver ought to permute rows and columns of **A** in order to preserve the sparsity during elimination, while on the other hand, it should use permutations for pivoting, i.e. to guard against division by zero [7]. This procedure always represents a compromise. If the solver concentrates too much on preserving sparsity, the

numerical properties of the Gaussian elimination may be compromised. On the other hand, if the solver ignores the sparsity in order to keep the values of the pivots large, this will lead to an increase in non–zero elements in the matrix, which in turn leads to a (possibly avoidable) overhead down the road. A general–purpose sparse–matrix solver that does a good job at balancing sparsity against a deterioration of the numerical condition is e.g. *subroutine HA28* from the Harwell library.

If additional knowledge about the system or about the structure of the equations is known, the solution can often be accomplished in a much more efficient way. For example, if $\mathbf{A}$ is symmetric and positive definite, it is well–known that pivoting is not necessary when applying a Cholesky decomposition, i.e., when transforming $\mathbf{A}$ to $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is a lower triangular matrix. This in turn leads to the desirable property that row and column permutations, in order to preserve the sparsity of $\mathbf{A}$ during elimination, can be done without knowing the numeric values of the parameters. Only the zero/non–zero pattern of $\mathbf{A}$ must be provided for this purpose. As a consequence, such a sorting algorithm can be applied once, before the start of a simulation, rather than repeating the procedure over and over again during each simulation step. At every time step, the elimination of $\mathbf{A}$ follows a fixed pattern that was determined before the simulation started.

In an object–oriented modeling system, it is difficult to use special–purpose sparse matrix solvers, such as the ones for banded systems or for positive definite systems. The reason is that the modeling system would have to deduce knowledge about the overall system structure. Only local properties of objects and their interconnections are known. Hence the only reasonable alternative seems to be to use a general–purpose sparse matrix solver. However, such a solution may be quite inefficient compared to the specialized algorithms that a domain–specific simulation program can use.

In this paper, it shall be shown that it is possible to introduce domain–specific knowledge in component model libraries of a general–purpose object–oriented modeling system. The symbolic manipulation algorithms inside the modeling software can then exploit this knowledge and generate code that is as efficient as the code that a special–purpose modeling system would generate. In this way, the object–oriented modeling system is able to solve even large systems of equations much more efficiently than if a general–purpose sparse–matrix solver would have been used.

In [10], a method called *tearing* was introduced to transform large sparse systems of equations down to small dense systems. The method was further elaborated in [11]. Below, a new complementary method, called *relaxing*, will be discussed that allows the modeler to provide more model knowledge to be used in the code optimization process within the modeling software.

2. BLT–TRANSFORMATION

In order to be able to discuss the *relaxing* method, the basic transformation algorithm of object–oriented modeling languages has to be reviewed. In general, a high–level, object–oriented model description leads directly to a large, sparse, non–linear system of equations that has to be solved for the unknown variables $\mathbf{z}$:

$$\mathbf{h}(\mathbf{z}) = \mathbf{0} \tag{2}$$

Usually, the first derivatives of the state variables, $\dot{\mathbf{x}}$, are elements of the vector $\mathbf{z}$. By permutation of equations and variables, it is possible to transform this system of equations to block–lower–triangular (BLT) form that can be solved in a nearly explicit forward sequence. The basic idea is explained by means of the following simple example consisting of three nonlinear equations:

$$\begin{array}{lcl} h_1(z_1, z_3) & = & 0 \\ h_2(z_2) & = & 0 \\ h_3(z_1, z_2) & = & 0 \end{array} \qquad \mathbf{S}_1 = \begin{array}{c} \begin{array}{ccc} z_1 & z_2 & z_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{array}\right] \end{array}$$

The structure of the system of equations is described by the *structure incidence matrix*, $\mathbf{S}$, displayed to the right of the equations. This matrix signals whether the k^{th} variable (k^{th} column) occurs in the i^{th} equation (i^{th} row), or not. By permuting equations and variables, this set of equations can be brought to BLT–form:

$$\begin{array}{lcl} h_2(z_2) & = & 0 \\ h_3(z_1, z_2) & = & 0 \\ h_1(z_1, z_3) & = & 0 \end{array} \qquad \mathbf{S}_2 = \begin{array}{c} \begin{array}{ccc} z_2 & z_1 & z_3 \end{array} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{array}\right] \end{array}$$

This process is also called *partitioning* the set of equations. The strictly lower–triangular form of the permuted structure incidence matrix characterizes the fact that the non-linear equations can be solved one at a time in a given sequence. First z_2 is evaluated from h_2, then h_3 is solved for z_1, and finally, z_3 is determined from h_1. If the variable to be solved for appears linearly in an equation, that equation can be transformed to explicit form by simple formulae manipulation. Otherwise, a local Newton iteration is needed.

In general, it is not possible to transform the structure incidence matrix to a strictly lower–triangular

form. However, efficient algorithms exist to transform to block–lower–triangular form, i.e., to a form, in which some blocks of dimensions ≥ 1 are present along the diagonal. The algorithm guarantees that the dimensions of the diagonal blocks are kept as small as possible, i.e., it is not possible to transform to blocks of yet smaller dimensions just by permuting variables and equations. Non–trivial blocks on the diagonal correspond to systems of equations that have to be solved simultaneously. In other words, the partitioning algorithm finds algebraic loops of minimal dimensions. Algorithmic details and a proof of the mentioned property can be found in [7].

3. AN INTRODUCTORY EXAMPLE

The simple drive train in figure 1 is used as an introductory example for the relaxing method. More advanced systems shall be treated in due course. Figure 1 shows a window from Dymola's object–diagram editor. The drive train consists of the rotor, `s1`, of a motor, which drives an ideal inertialess gear, `g`, which in turn drives the load inertia, `s2`.

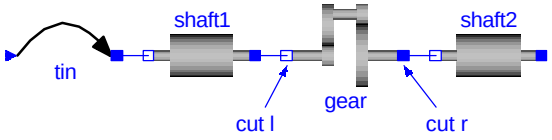

Figure 1: Object diagram of a drive train

In Dymola the components of a drive train can be described in the following way:

```
model class Shaft
  parameter J
  cut l (w, a / tl)  {left cut}
  cut r (w, a / - tr)  {right cut}

    J * relax(a) = tl - tr
end

model class ShaftS
  parameter J
  cut l (w, a / tl)
  cut r (w, a / - tr)

    der(w) = a
    J * a = tl - tr
end

model class Gear
  parameter i = 1
  cut l (wl, al / tl)
  cut r (wr, ar / - tr)

    wl = i * wr
    al = i * relax(ar)
    i * tl = tr
end
```

Each component consists of two cuts, with which the component can be rigidly connected to other drive train components. The variables in the cut definitions are the angular velocity, `w`, the angular acceleration, `a`, and the cut–torque, `t`. Cut–variables at the right–hand side of the slash "/" delimiter are *through variables*, i.e., when connected at a point, these variables are summed up to zero. The reader is asked to ignore the "compiler–directive" `relax` for the time being. Class `Shaft` describes shafts with two cuts. The single equation states a torque balance. Class `ShaftS` is a shaft where the angular velocity is used as a state variable. Finally, class `Gear` contains the equations of an ideal gear, in which the angular velocity and acceleration are transformed via the gear ratio, `i`, whereas the torques are transformed via the inverse relationship. After adding graphical information, the components can be connected together at cuts as shown in figure 1.

In order to produce a state–space description of this system, the model compiler first collects the local equations of all components, adds the equations according to the connection structure, and transforms the resulting system of equations to BLT–form. Using the abbreviations, α_1 = `shaft1.a`, α_2 = `shaft2.a`, τ_1 = `shaft1.tl`, and τ_2 = `shaft2.tr`, the BLT–form of the drive train is given by:

$$\begin{bmatrix} J_1 & 0 & 1 & 0 \\ 0 & J_2 & 0 & -1 \\ -1 & i & 0 & 0 \\ 0 & 0 & i & -1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \tau_1 \\ \tau_2 \end{bmatrix} = \begin{bmatrix} \tau_{in} \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (3)$$

$$\dot{\omega}_2 = \alpha_2 \quad (4)$$

In other words, the BLT–form of this example consists of primarily one single block. Note that it is not possible to reduce this system of equations further by mere permutation of rows and columns of the matrix. Furthermore note that the ordering of the rows within a diagonal block of the BLT–form is not unique, i.e., the system of equations above could also have been presented in another sequence. In order to compute the derivative of the state variable, i.e., $\dot{\omega}_2$, a system of 4 linear equations has to be solved at every step of the simulation.

When the reader would solve this problem by hand, he or she could do much better. Since the system matrix in (3) contains many zero and one elements, it is easy to eliminate variables, and to end up with the well–known equation:

$$(J_2 + i^2 J_1)\,\dot{\omega}_2 = i\tau_{in} \quad (5)$$

that can easily be solved for the only remaining unknown variable, $\dot{\omega}_2$.

The BLT–transformation performs generally not as poorly as this example seems to indicate. Assume that additional drive train components are attached via rotational springs to the drive train above (describing the gear elasticity). In this case, the BLT–transformation will detect isolated algebraic loops, one for each part, which consist of rigidly attached shafts and gear boxes only, i.e., a spring acts as a separator between algebraic loops.

4. RELAXING

Assume that the following linear system of equations should be solved:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{A}_{13} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \mathbf{b}_3 \end{bmatrix} \quad (6)$$

where the $\mathbf{A}_{ij}$ are block matrices of appropriate dimensions. When using standard Gaussian elimination, the sequence of the elimination process depends on the numerical values of the matrix elements. *Relaxing* is a way how this sequence can be defined by the modeler:

1. Characterize some of the unknown variables in some equations by $\mathbf{relax}(\mathbf{x}_i)$.
2. Transform the system to BLT–form assuming that the "relaxed" variables are not present. To distinguish from the "usual" BLT–transformation, this partitioning is called *relaxed BLT–form*. The relaxed BLT–form will have diagonal blocks of smaller dimensions.
3. Solve the system in BLT–form by Gaussian elimination in such an order that the variables of the diagonal blocks of the BLT–form are eliminated in sequence. Since the sequence is fixed, the elimination can be done *symbolically*, thereby utilizing the zero/non–zero pattern of the matrix.

Equation (6) could be relaxed in the following way:

$$\begin{aligned} \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{relax}(\mathbf{x}_2) + \mathbf{A}_{13}\mathbf{relax}(\mathbf{x}_3) &= \mathbf{b}_1 \\ \mathbf{A}_{21}\mathbf{x}_1 + \mathbf{A}_{22}\mathbf{x}_2 \quad\quad + \mathbf{A}_{23}\mathbf{relax}(\mathbf{x}_3) &= \mathbf{b}_2 \\ \mathbf{A}_{31}\mathbf{x}_1 + \mathbf{A}_{32}\mathbf{x}_2 \quad\quad + \mathbf{A}_{33}\mathbf{x}_3 \quad\quad &= \mathbf{b}_3 \end{aligned} \quad (7)$$

i.e., during partitioning it is assumed that the first equation is a function of $\mathbf{x}_1$ only. The relaxed BLT–form reduces the previously full–sized matrix to block diagonal form:

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{0} \\ \mathbf{A}_{31} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 - \mathbf{A}_{12}\mathbf{relax}(\mathbf{x}_2) - \mathbf{A}_{13}\mathbf{relax}(\mathbf{x}_3) \\ \mathbf{b}_2 - \mathbf{A}_{23}\mathbf{relax}(\mathbf{x}_3) \\ \mathbf{b}_3 \end{bmatrix}$$

Gaussian elimination in the indicated sequence means that first $\mathbf{x}_1$ is determined as a function of $\mathbf{relax}(\mathbf{x}_2), \mathbf{relax}(\mathbf{x}_3)$:

$$\mathbf{x}_1 := \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{relax}(\mathbf{x}_2) - \mathbf{A}_{13}\mathbf{relax}(\mathbf{x}_3)) \quad (8)$$

Inserting this relationship into the second and third equations of (7) results in a reduced set of equations:

$$\begin{aligned} \mathbf{A}_{22}^{(2)}\mathbf{x}_2 + \mathbf{A}_{23}^{(2)}\mathbf{relax}(\mathbf{x}_3) &= \mathbf{b}_2^{(2)} \\ \mathbf{A}_{32}^{(2)}\mathbf{x}_2 + \mathbf{A}_{33}^{(2)}\mathbf{x}_3 &= \mathbf{b}_3^{(2)} \end{aligned} \quad (9)$$

where:

$$\begin{aligned} \mathbf{A}_{ij}^{(1)} &:= \mathbf{A}_{ij} \\ \mathbf{b}_i^{(1)} &:= \mathbf{b}_i \\ \mathbf{A}_{ij}^{(k+1)} &:= \mathbf{A}_{ij}^{(k)} - \mathbf{A}_{ik}^{(k)}\mathbf{A}_{kk}^{(k)^{-1}}\mathbf{A}_{kj}^{(k)} \\ \mathbf{b}_i^{(k+1)} &:= \mathbf{b}_i^{(k)} - \mathbf{A}_{ik}^{(k)}\mathbf{A}_{kk}^{(k)^{-1}}\mathbf{b}_k^{(k)} \end{aligned}$$

Solving the first equation in (9) for $\mathbf{x}_2$ results in:

$$\mathbf{x}_2 := \mathbf{A}_{22}^{(2)^{-1}}(\mathbf{b}_2^{(2)} - \mathbf{A}_{23}^{(2)}\mathbf{relax}(\mathbf{x}_3)) \quad (10)$$

Inserting this relationship again into the last row of (9) finally gives an equation that can explicitly be solved for $\mathbf{x}_3$:

$$\mathbf{A}_{33}^{(3)}\mathbf{x}_3 = \mathbf{b}_3^{(3)} \quad (11)$$

After $\mathbf{x}_3$ is computed, $\mathbf{x}_2$ can be obtained from (10) and finally $\mathbf{x}_1$ from (8), or in general:

$$\mathbf{x}_k := \mathbf{A}_{kk}^{(k)^{-1}}(\mathbf{b}_k^{(k)} - \sum_{j=k+1}^{n} \mathbf{A}_{kj}^{(k)}\mathbf{x}_j)$$

In order for relaxing to be possible, two conditions must be fulfilled. First, the relaxed variables must be chosen in such a way that a transformation to BLT–form is possible, i.e., that the system matrix is not structurally singular. This would not be the case if e.g. variable $\mathbf{x}_2$ in the second row of (7) would also be relaxed. Second, the Gaussian elimination must not lead to a division by zero. The first condition can be checked automatically by the model compiler. However, it is not possible to check the second condition without knowing the actual numerical values of the matrix elements.

Relaxing has the advantage that the sequence of computation can be determined before a simulation run starts, thereby utilizing the sparsity of the matrix. For example, if one of the block matrices $\mathbf{A}_{ij}$ is zero, all terms in which this matrix appears can be canceled. By appropriately chosen relaxing variables, newly computed intermediate matrices $\mathbf{A}_{ij}^{(k)}$ remain zero too, so that the original sparsity pattern is kept as much as possible also for the intermediate, smaller systems of equations. In particular, the computation can be done

in a fully symbolical way, which would not otherwise be possible[1]. Relaxing has the drawback that it can be difficult for the modeler to decide how a system of equations should be relaxed correctly. Especially, it must be guaranteed that the predefined sequence does not lead to division by zero.

Above, relaxing was used to directly solve the given linear system of equations by a specific sequence in Gaussian elimination. Alternatively, the relaxing definition can be utilized to solve the linear (or even a non–linear) system of equations iteratively, which can be useful for very large systems of equations (say > 10000 equations). In such a case, the relaxed variables can be used as *iteration variables* in a fixed–point iteration. For equation (7), this results in the following iteration scheme:

$$
\begin{aligned}
&\mathbf{x}_2 := \text{init}(\mathbf{x}_2)\\
&\mathbf{x}_3 := \text{init}(\mathbf{x}_3)\\
&\textbf{repeat}\\
&\quad \textbf{relax}(\mathbf{x}_2) := \mathbf{x}_2\\
&\quad \textbf{relax}(\mathbf{x}_3) := \mathbf{x}_3\\
&\quad \mathbf{x}_1 := \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\textbf{relax}(\mathbf{x}_2)\\
&\qquad\qquad - \mathbf{A}_{13}\textbf{relax}(\mathbf{x}_3))\\
&\quad \mathbf{x}_2 := \mathbf{A}_{22}^{-1}(\mathbf{b}_2 - \mathbf{A}_{21}\mathbf{x}_1 - \mathbf{A}_{23}\textbf{relax}(\mathbf{x}_3))\\
&\quad \mathbf{x}_3 := \mathbf{A}_{33}^{-1}(\mathbf{b}_3 - \mathbf{A}_{31}\mathbf{x}_1 - \mathbf{A}_{32}\mathbf{x}_2)\\
&\textbf{until}\ \text{converged}(\mathbf{x}_2 - \textbf{relax}(\mathbf{x}_2))\ \textbf{and}\\
&\qquad \text{converged}(\mathbf{x}_3 - \textbf{relax}(\mathbf{x}_3))
\end{aligned}
$$

The relaxing iteration scheme corresponds essentially to a Gauss–Seidel iteration. However, the symbolic implementation of this algorithm within an object–oriented modeling system provides a much larger degree of flexibilty than any numerical implementation of the algorithm ever would.

1. A standard Gauss–Seidel iteration uses all of the unknown variables as iteration variables. In a relaxing iteration scheme, this is seldomly the case. For example, if matrix $\mathbf{A}_{21}$ were to be zero in the above example, only $\mathbf{x}_3$ would have to be used as an iteration variable, which enhances both the stability and efficiency of the iteration.

2. A standard Gauss–Seidel iteration works on scalar equations. In a relaxing iteration scheme, it is easy to work additionally on block matrices, and to solve the small blocks $\mathbf{A}_{ii}$ by Gaussian elimination. For some types of systems, this enhances the stability of the algorithm further.

3. The set–up of an iteration is very easy in an object–oriented modeling systems such as Dymola. The **relax** operator must simply be supplied in the **class** definitions at appropriate places. If the model is build up by objects of the same class, which is e.g. the case when discretizing partial differential equations, it suffices to place the **relax** operator at a single location in order to get a suitable iteration scheme.

[1] Using determinants, it is always possible to solve a system of equations symbolically. However, this method is extremely inefficient for medium to large system sizes.

In [10], several alternatives of tearing are discussed. Especially, branch and node tearing in the bipartite graph of a system of equations are mentioned. The paper explains in detail how *node tearing* can be used in an object–oriented modeling system. *Relaxing* can be seen as an application of the *branch tearing* method.

5. RELAXING DRIVE TRAINS

The drive train library presented earlier in this paper already contains appropriate relaxing information. For the drive train example in figure 1, this leads to the following relaxed BLT–form:

$$
\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ i & 0 & -1 & 0 \\ 0 & 0 & -1 & J_2 \end{bmatrix}
\begin{bmatrix} \tau_1 \\ \alpha_1 \\ \tau_2 \\ \alpha_2 \end{bmatrix}
=
\begin{bmatrix} -\tau_{in} + J_1\textbf{relax}(\alpha_1) \\ i \cdot \textbf{relax}(\alpha_2) \\ 0 \\ 0 \end{bmatrix}
\tag{12}
$$

Applying Gaussian elimination finally results in:

$$
\begin{aligned}
c_1 &:= -i\tau_{in}\\
c_2 &:= -iJ_1\\
c_3 &:= ic_2\\
c_4 &:= J_2 - c_3\\
\alpha_2 &:= -c_1/c_4\\
\tau_2 &:= -c_1 + c_3\alpha_2\\
\alpha_1 &:= i\alpha_2\\
\tau_1 &:= \tau_{in} - J_1\alpha_1
\end{aligned}
$$

As can be seen, this implementation is equivalent to the hand–derived equation (5) and much more efficient than to repetitively solve a linear system of four equations.

At first sight, it seems rather difficult to apply the relaxing technique. Where should the relax operators be placed in the models? Looking at equation (8) may provide some hints as to how the relaxing variables ought to be selected. Whenever the modeler knows that an equation in a model class should be solved for a specific variable and it turns out that this equation usually ends up in algebraic loops, all the remaining variables in this equation should be relaxed. This rule of thumb can be applied to a drive train.

A drive train can be split into "parts," where each part consists of rigidly attached shafts and gears, and where the parts are connected to each other via "force"

elements such as springs or dampers. Each part is described by one state variable, the angular velocity of one of its components. Using all state variables, the angular velocities of all components of a part as well as the relative angular velocities between the parts can be determined. Since force laws are functions of the state variables, the torques produced by all force laws can be computed as well. At this stage, the drive train parts are totally decoupled from each other, since the torques at both ends of all parts are known. Traversing recursively from both ends of a part in the direction of the element defining the state variable, there is always the situation, that the torque at one side of a component is known, and that the torque at the other side of the component should be computed. If this could be done, all torques in a part could be calculated. In order to be able to proceed in this way, the other unknown variables in the equations, i.e., the angular accelerations, have to be relaxed.

6. RELAXING ELECTRICAL CIRCUITS

Some basic electrical elements of an electrical circuit can be described by the following model classes:

```
model class OnePort
   {superclass of all one – –port elements}
   cut p ( Vp / i )  {positive pin}
   cut n ( Vn / – i )  {negative pin}
   local u {voltage drop}
      u = Vp – Vn
end

model class Resistor
   inherit OnePort
   parameter R {Resistance}
      R * i = Vp – Vn
end

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

Other elements, like inductors and voltage or current sources, are described in similar ways. When electrical elements are connected together, a large, sparse system of differential and algebraic variables occurs that is characterized by the following structure:

$$\mathbf{0} = \mathbf{A}\mathbf{i}(t) \quad (13)$$

$$\mathbf{u}(t) = \mathbf{A}^T\mathbf{v}(t) \quad (14)$$

$$\mathbf{i}_v(t) = \mathbf{h}_v(\dot{\mathbf{u}}(t), \mathbf{u}(t), \mathbf{v}(t)) \quad (15)$$

$$\mathbf{0} = \mathbf{h}_r(\dot{\mathbf{u}}(t), \mathbf{u}(t), \mathbf{v}(t), d\mathbf{i}_r(t)/dt, \mathbf{i}_r(t)) \quad (16)$$

$$\mathbf{i} = [\mathbf{i}_c; \mathbf{i}_r] \quad (17)$$

where $\mathbf{A}$ is the *reduced incidence matrix* describing the connection structure of the electrical elements[2]. (13) states that the sum of the currents $\mathbf{i}$ at every node is zero, (14) computes the voltage drop $\mathbf{u}$ across all elements from the node potentials $\mathbf{v}$, (15) are the component equations of voltage–controlled elements (such as resistors and capacitors), and (16) are the component equations of the remaining elements (such as inductors). (13–16) are called the *Sparse Tableau Equations* [13, 20, 19]. Note that this system of equations contains the potentials v_i of all nodes, the potential drops u_i through all elements[3], and the currents i_k through all voltage–controlled elements as unknown variables.

Experience shows that the BLT–transformation often ends up rather quickly in one very large block that cannot be further partitioned. With the class definitions as given above, the system of equations (13-16) is already reduced, because the voltage drops (14) are inserted into the element equations (15, 16) (with the exception of voltage drops used as state variables), i.e., these variables are eliminated from the system of equations. The sparse tableau equations are seldom directly used by the sparse matrix solvers employed in electric circuit programs. Instead, programs like SPICE [20] and Saber [18] first eliminate the currents $\mathbf{i}_v$ of all voltage–controlled elements. This method is called *Modified Nodal Analysis (MNA)* [14, 20, 19]. The two popular methods, cutset analysis and loop analysis, often found in introductory books on network analysis, impose restrictions upon the types of branch relations that can be present in a network, and are therefore not used in general–purpose circuit simulation programs.

Using the relaxing method, it is easy for an object–oriented modeling system to follow the MNA–philosophy. To this end, it suffices to relax the unknown potentials in the voltage–controlled elements:

```
model class Resistor
   inherit OnePort
   parameter R {Resistance}
      R * i = relax(Vp) – relax(Vn)
end

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

As a consequence, the component equations of voltage–controlled elements are solved for the currents $\mathbf{i}_v$ during relaxed BLT–transformation. These currents

[2] A_{ij} is one when the j–th branch leaves the i–th node, it is minus one, when the branch arrives at the node, and it is zero otherwise.

[3] with the exception of u_j used as state variables, as in capacitors.

are in turn inserted into equation (13) by Gaussian elimination, i.e., equations (15) are completely eliminated from the systems of equations, and the currents $\mathbf{i}_v$ are removed from the vector of unknown variables. This procedure is demonstrated by means of an excerpt of a circuit as shown in figure 2. Expanding the

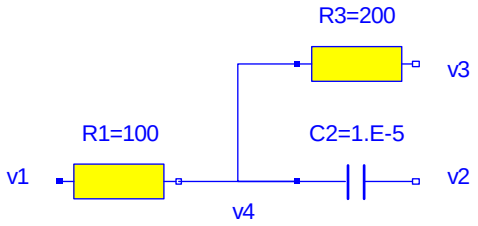

Figure 2: Relaxing an electrical circuit

equations from the model classes results in the following relaxed BLT–form:

$$\begin{aligned} i_2 &= C\mathbf{relax}(\dot{u}_2) \\ R_1 i_1 &= \mathbf{relax}(v_1) - \mathbf{relax}(v_4) \\ R_3 i_3 &= \mathbf{relax}(v_3) - \mathbf{relax}(v_4) \\ v_2 - v_4 &= u_2 \\ i_1 + i_2 + i_3 &= 0 \end{aligned}$$

Substituting the relaxed variables reduces the system of equations from five down to two equations:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & -1 \\ C & \frac{1}{R_1} & 0 & \frac{1}{R_3} & -\frac{1}{R_1} - \frac{1}{R_3} \end{bmatrix} \begin{bmatrix} \dot{u} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} u \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Although relaxing has already drastically reduced the system size, the remaining system is still sparse, and a general–purpose sparse matrix solver is needed to solve it efficiently. Note that the modeler using the circuit elements presented above does not have to know anything about the relaxing method. He or she only needs to invoke component models from the electrical component library, and connect them together. The information about the solution strategy is hidden in the library model classes.

7. TEARING

In the next section, the relaxing of three–dimensional multibody systems is discussed, which needs the tearing method for a subproblem. Tearing is therefore briefly reviewed. Tearing is a well–known method for solving sets of algebraically coupled equations. It had originally been introduced by Kron [15]. In [10], it has been shown how physical insight can extend the applicability of this method, and it was demonstrated that tearing is particularly useful in object–oriented modeling systems. Tearing of a linear or non–linear system:

$$\mathbf{h}(\mathbf{x}) = \mathbf{0} \tag{18}$$

means:

1. to split the vector of unknown variables into two parts called the *tearing variables*, $\mathbf{x}_t$, and the *remaining variables*, $\mathbf{x}_r$, and
2. to select some equations of $\mathbf{h}$ as *residue equations* in such a way that the residues can be determined, provided the tearing variables are known.

In order to define tearing variables and residue equations in an unambiguous way, the following notation is used:

$$\mathbf{h}_1(\mathbf{x}_r, \mathbf{x}_t) = \mathbf{0} \tag{19}$$

$$\mathbf{h}_2(\mathbf{x}_r, \mathbf{x}_t) = \mathbf{residue}(\mathbf{x}_t) \tag{20}$$

meaning that the residues can be calculated, provided the tearing variables, $\mathbf{x}_t$, are known. This requires a specific structure of equation, $\mathbf{h}_1$. Tearing can be quite effective, because a (usually sparse) system of dimension $dim(\mathbf{h}_1) + dim(\mathbf{h}_2)$ is reduced to a (usually full) system of dimension $dim(\mathbf{h}_2)$, which is then solved by a standard, general–purpose, linear or non–linear solver (the solver provides $\mathbf{x}_t$ and gets the residue). In the linear case, equations (19,20) have the following structure:

$$\begin{bmatrix} \mathbf{L} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_r \\ \mathbf{x}_t \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 + \mathbf{residue}(\mathbf{x}_t) \end{bmatrix} \tag{21}$$

where $\mathbf{L}$ is required to be a lower–triangular, non–singular matrix. In this case, the linear system of equations can be transformed to a linear system of smaller dimension:

$$(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{L}^{-1}\mathbf{A}_{12})\mathbf{x}_t = \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{L}^{-1}\mathbf{b}_1 \tag{22}$$

Since $\mathbf{L}$ is lower triangular, the inversion of $\mathbf{L}$ is just a backward substitution, which can be carried out in a fully symbolic fashion. The transformation, both for linear and non–linear tearing, is a special application of the BLT–transformation, as shown in [10], and can therefore be accomplished efficiently. The difficulty of using the tearing method is related to the selection of tearing variables and residue equations. In [10], it is shown, how physical insight often leads naturally to an appropriate choice.

8. TEARING MULTIBODY SYSTEMS

Multibody systems consist of rigid bodies that are connected together via ideal joints and force elements. For simplicity, the following section assumes that only tree–structured multibody systems are considered, i.e., the connection structure of bodies and joints does not contain closed kinematic loops. However, the results can be extended to closed–loop systems as well.

The components of a multibody system are treated as elements with one or two mechanical cuts, where the components can be rigidly connected together. Among other variables, the angular accelerations, $\boldsymbol{\alpha}_a$, and linear accelerations, $\mathbf{a}_a$, of cut a, as well as the cut–torques, $\boldsymbol{\tau}_a$, and cut–forces, $\mathbf{f}_a$, are transported through a cut. For compactness, these variables are collected together:

$$\hat{\mathbf{a}}_a = \begin{bmatrix} \boldsymbol{\alpha}_a \\ \mathbf{a}_a \end{bmatrix} \quad \hat{\mathbf{f}}_a = \begin{bmatrix} \boldsymbol{\tau}_a \\ \mathbf{f}_a \end{bmatrix} \tag{23}$$

For simplicity, it is assumed that all vectors are resolved in a common frame. Multibody systems consist of rigid bodies, bars, and ideal joints. Since force elements usually don't pose difficulties in the BLT–transformation, they are not discussed here. A *rigid body* has only one cut at its center of mass, and is described by Newton's and Euler's equations:

$$\mathbf{I}_a \hat{\mathbf{a}}_a \;=\; \hat{\mathbf{f}}_a + \mathbf{b}_a \tag{24}$$

where:

$$\mathbf{I}_a = \begin{bmatrix} \mathbf{I}^{CM} & \mathbf{0} \\ \mathbf{0} & m\mathbf{E} \end{bmatrix} \quad \mathbf{b}_a = \begin{bmatrix} -\boldsymbol{\omega}_a \times \mathbf{I}^{CM}\boldsymbol{\omega}_a \\ \mathbf{0} \end{bmatrix}$$

$\mathbf{I}^{CM}$ is the inertia tensor with respect to the center of mass, m is the mass of the body, $\mathbf{E}$ is a unit matrix, and $\boldsymbol{\omega}_a$ is the angular velocity of the body. A *bar* is a massless mechanical element with two mechanical cuts described by the following equations (the equations on position and velocity level are not given due to space limitations):

$$\hat{\mathbf{a}}_b \;=\; \mathbf{C}_b \mathbf{a}_a + \boldsymbol{\zeta}_b \tag{25}$$

$$\hat{\mathbf{f}}_a \;=\; \mathbf{C}_b^T \hat{\mathbf{f}}_b \tag{26}$$

with:

$$\mathbf{C}_b \;=\; \begin{bmatrix} \mathbf{E} & \mathbf{0} \\ -\mathbf{skew}(\mathbf{r}^{ab}) & \mathbf{E} \end{bmatrix}$$

$$\boldsymbol{\zeta}_b \;=\; \begin{bmatrix} \mathbf{0} \\ \boldsymbol{\omega}_a \times (\boldsymbol{\omega}_a \times \mathbf{r}^{ab}) \end{bmatrix}$$

$$\mathbf{skew}(r) \;=\; \begin{bmatrix} 0 & -r_3 & r_2 \\ r_3 & 0 & -r_1 \\ -r_2 & r_1 & 0 \end{bmatrix}$$

where $\mathbf{r}^{ab}$ is the position vector from cut a to cut b of the bar. The equations state that the accelerations at cut b can be computed from the accelerations at cut a, and that the cut–forces at cut a can be computed from the cut–forces at cut b, due to a force/torque balance at the massless element.

Finally, joints are massless elements with two cuts. Contrary to the bar, the movement of cut b with respect to cut a is not fully restricted. Instead, it is described by f_i generalized coordinates $\mathbf{q}_i$ (a bar could be seen as a joint with $f_i = 0$). A typical representation of a joint is a revolute joint. On acceleration level, it is described by the following equations:

$$\hat{\mathbf{a}}_b \;=\; \mathbf{relax}(\hat{\mathbf{a}}_a) + \hat{\mathbf{n}}\ddot{q} + \boldsymbol{\zeta}_b \tag{27}$$

$$\hat{\mathbf{f}}_a \;=\; \hat{\mathbf{f}}_b \tag{28}$$

$$\hat{\mathbf{n}}^T \hat{\mathbf{f}}_b \;=\; \tau + \mathbf{residue}(\ddot{q}) \tag{29}$$

where:

$$\boldsymbol{\zeta}_b = \begin{bmatrix} \boldsymbol{\omega}_a \times \mathbf{n}\dot{q} \\ \mathbf{0} \end{bmatrix} \quad \hat{\mathbf{n}} = \begin{bmatrix} \mathbf{n} \\ \mathbf{0} \end{bmatrix}$$

$\mathbf{n}$ is a unit vector in the direction of the rotation axis, and q is the rotation angle. The third equation is due to d'Alemberts principle (preserving energy flow), and states that the projection of the cut–force onto the axis of rotation is equal to the torque τ driving the rotation axis (τ may be zero).

The reader is asked to ignore the **relax** operator and the term $\mathbf{residue}(\ddot{q})$ for the moment. For tree–structured systems, it is best to use the generalized coordinates of all joints as state variables. In the case of a revolute joint, these are the angle of rotation, q, and the relative angular velocity, $\dot{q}$.

A multibody model is built up by connecting together specific objects of the discussed types, cf. e.g. figure 3. For most models, BLT–transformation will result in one very large system of differential and algebraic equations containing the accelerations, $\hat{\mathbf{a}}$, and cut–forces/torques, $\hat{\mathbf{f}}$, at all cuts, as well as the generalized accelerations, $\ddot{\mathbf{q}}$, of all joints. Even for a modest–sized multibody system, such as a six degree of freedom robot, this leads already to a sparsely populated system of more than 200 equations.

In multibody dynamics, it is well–known that the equations can be transformed to the standard form:

$$\mathbf{M}(\mathbf{q}, t)\ddot{\mathbf{q}} \;=\; \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}, t) \tag{30}$$

In the case of a robot, these are six equations. In [10], it was shown how tearing can produce this result, too. In order to arrive at (30), it is clear that the $\ddot{\mathbf{q}}$ must be used as tearing variables, since the reduced system of equations (22) must have the same unknown variables as equation (30). The selection of the residue equations is more difficult, and is explained in [10]. In short, it is well known in robotics that all desired quantities can

be computed, provided $\mathbf{q}$, $\dot{\mathbf{q}}$, and $\ddot{\mathbf{q}}$ are given. Especially, the driving torques of the joints can be calculated (the so–called inverse dynamics problem), which justifies the selection of $\ddot{\mathbf{q}}$ as tearing variables. Introducing the **residue** operator at the equation of the driving torques (29) will produce a system structure of (21) with a lower–triangular, non–singular $\mathbf{L}$ matrix. As explained in the last section, this system can be transformed to (22), i.e., to equation (30).

As can be seen, even handling 3D multibody systems becomes surprisingly easy compared to traditional approaches, if the object–oriented method is used. Only the local equations of the standard components have to be formulated (24–29). After introducing appropriate tearing information, a software tool, such as Dymola, can automatically transform to the standard form.

9. RELAXING MULTIBODY SYSTEMS

Matrix $\mathbf{M}$ in (30) has f^2 elements. It is therefore understandable that the most efficient methods to transform to (30) need $O(f^2)$ operations in order to compute $\mathbf{M}$, and $O(f^3)$ operations in order to solve the linear system of equations (30) for the unknown variables, $\ddot{\mathbf{q}}$.

In the eighties, a new class of multibody algorithms, the so–called *recursive O(f) algorithms*, appeared that can solve for $\ddot{\mathbf{q}}$ in $O(f)$ operations, if the multibody system has a tree structure [3, 12, 5, 4]. Generalizations to closed–loop systems are also available, but are not considered here for simplicity.

Modern commercial multibody programs, such as SIMPACK [22] or DADS [23], utilize one of the variants of this algorithm class. Clearly for larger systems, the $O(f)$ property is highly desirable. However even for small systems, such as systems with two degrees of freedom, an *appropriately implemented* recursive algorithm is at least as efficient as one of the $O(f^3)$ methods [5].

It turns out that the relaxing method is powerful enough to reproduce an $O(f)$ algorithm in a fully automated fashion. The library developer only needs to introduce the **relax** operator into the joint classes in the way shown in equation (27). It is really an astonishing result that such a special multibody algorithm can be realized within Dymola by just inserting the **relax** keyword at a few places. It seems not possible to implement such an algorithm in an easier way.

It is beyond the scope of this paper to rigidly prove that the relaxing method will actually generate one of the variants of the recursive $O(f)$ algorithms. Although the result is very simple, the proof is lengthy. Instead, only one part of the proof will be given that gives at the same time more insight why relaxing will work so satisfactorily in this case. As explained in the section entitled *RELAXING* of this paper, a necessary condition for relaxing is, that the relaxed BLT–transformation can be carried out, i.e., that the system with the removed relaxing variables is not structurally singular. It will now be shown that this condition is fulfilled for selection (27):

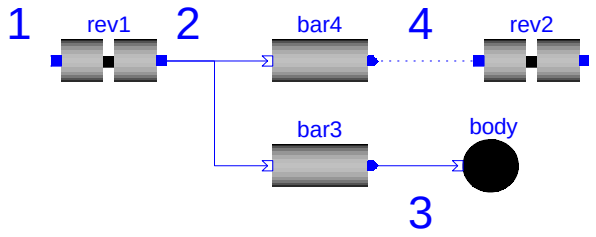

Figure 3: Object diagram of multibody system

In figure 3, a typical part of a multibody system is shown that is built up from basic components. It consists of a *joint* that is rigidly connected at location 2 with two *bars*, called bar 3 and bar 4. The other side of bar 3 ends at the center of mass of a body, and is rigidly connected with that body object. Bar 4 is the connecting bar between joint 1 and the subsequent joint. The accelerations and constraint forces at location i in the figure are denoted as $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{f}}_i$, respectively. According to (24–29), this subsystem of the overall multibody model is described by the following equations:

$$\begin{bmatrix} \mathbf{I}_3 & -\mathbf{E} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ -\mathbf{E} & \mathbf{0} & \mathbf{C}_3 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{C}_3^T & \mathbf{0} & -\mathbf{E} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\mathbf{E} & \mathbf{0} & \hat{\mathbf{n}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \hat{\mathbf{n}}^T & \mathbf{0} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{a}}_3 \\ \hat{\mathbf{f}}_3 \\ \hat{\mathbf{a}}_2 \\ \hat{\mathbf{f}}_2 \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_3 \\ -\boldsymbol{\zeta}_3 \\ -\mathbf{C}_4^T \hat{\mathbf{f}}_4 \\ -\mathbf{C}_2 \mathbf{relax}(\hat{\mathbf{a}}_1) - \boldsymbol{\zeta}_2 \\ \tau + \mathbf{residue}(\ddot{q}) \end{bmatrix} \quad (31)$$

$$\hat{\mathbf{a}}_4 = \mathbf{C}_4 \hat{\mathbf{a}}_2 + \boldsymbol{\zeta}_4 \quad (32)$$

$$\hat{\mathbf{f}}_1 = \mathbf{C}_2^T \hat{\mathbf{f}}_2 \quad (33)$$

Transformation to relaxed BLT–form means that, in this phase of the analysis, the relaxed variables in the defined equations are assumed to be known. Note that the variables are *only* assumed to be known in the equations, where the relax–operator is used (this is in contrast to tearing, where the tearing variables are assumed to be known in *all* equations). The necessary condition is fulfilled, if it can be shown that the remaining system can be solved for all unknown variables.

The leaves of a tree–structured multibody system contain only bodies and no joints (otherwise the joints could be removed without any effect on the dynamics). The structure of a leaf of the tree is given in figure 3, with the only addition that bar 4 is not present, because a leaf ends in a body. This means that $\hat{\mathbf{f}}_4 = \mathbf{0}$ in (31). Since $\mathbf{relax}(\hat{\mathbf{a}}_1)$ is assumed to be known, all variables on the right–hand side are known, and (31) is a system of 25 equations for the 25 unknown variables $\hat{\mathbf{a}}_3, \hat{\mathbf{f}}_3, \hat{\mathbf{a}}_2, \hat{\mathbf{f}}_2, \ddot{q}$, which can be solved by Gaussian elimination (remember that each of the vectors has six elements). This system corresponds to a block–matrix $\mathbf{A}_{ii}$ in (7). After solving this system of equations, the unknown variables $\hat{\mathbf{a}}_4, \hat{\mathbf{f}}_1$ can be computed directly from (32,33).

The subcomponent, to which the leaf subcomponent is connected, has again the structure shown in figure 3. Now, bar 4 is present, and is connected with the joint of the leaf component. The leaf component can be removed, and can be replaced by $\hat{\mathbf{f}}_4$, which is known, because it was computed as $\hat{\mathbf{f}}_1$ in the leaf component. Therefore, the equations have exactly the same structure (31-33) as they had for the leaf component. As a consequence, all unknown variables of this system of equations can again be computed. By removing this subcomponent too, a backward recursion is established in a natural way. When the recursion stops at the root of the tree, i.e., at the inertial system, all unknown variables have been computed. In other words, the relaxed BLT–transformation is completed successfully. **q.e.d.**

The efficiency can be enhanced considerably in (31), if the structure of the sparse system of equations of the subblocks is utilized. It is not possible to use relaxing again: in order to transform (31) to relaxed BLT–form, the variables in the upper subdiagonal must be relaxed. However, if this were to be done, the remaining matrix would clearly be structurally singular due to the zero matrices on the diagonal. Another possibility would be to just relax $\ddot{\mathbf{q}}$ in row 4. However, this also leads to a structurally singular relaxed BLT–form.

A simple remedy is to use *tearing* with exactly the same tearing variables and residue equations that are utilized to transform into the standard form (30). The appropriate **residue** operator is already introduced in (31). Tearing will reduce the system of 25 equations to lower–triangular form, so that the inversion becomes trivial. It can be seen easily that tearing will work: assume that the tearing variable $\ddot{q}$ is known. Then $\hat{\mathbf{a}}_2$ can be computed from the fourth row of (31), $\hat{\mathbf{a}}_3$ from the second row, $\hat{\mathbf{f}}_3$ from the first row, $\hat{\mathbf{f}}_2$ from the third row, and finally, the residue from the last row. In other words, all unknown variables and the residue can be computed provided the tearing variable $\ddot{q}$ is known, i.e., the tearing is complete.

The tearing algorithm has to be modified, in order that it can be used to solve subproblems in a relaxed BLT–transformation. To this end, a third row has to be added to the system matrix of (21):

$$\begin{bmatrix} \mathbf{L} & \mathbf{A}_{12}\mathbf{A}_{13} \\ \mathbf{A}_{21}\mathbf{A}_{22}\mathbf{A}_{23} & \end{bmatrix} \begin{bmatrix} \mathbf{x}_r \\ \mathbf{x}_t \\ \mathbf{x}_{relax} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 + \mathbf{residue}(\mathbf{x}_t) \end{bmatrix} \tag{34}$$

containing the relaxed variables. When $\mathbf{L}$ is non–singular and lower–triangular, this system of equations can be transformed (in a fully symbolic fashion) to:

$$\mathbf{A}_{22}^{(2)}\mathbf{x}_t = \mathbf{b}_2^{(2)} - \mathbf{A}_{23}^{(2)}\mathbf{x}_{relax} \tag{35}$$

where

$$\begin{aligned} \mathbf{A}_{22}^{(2)} &= \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{L}^{-1}\mathbf{A}_{12} \\ \mathbf{A}_{23}^{(2)} &= \mathbf{A}_{23} - \mathbf{A}_{21}\mathbf{L}^{-1}\mathbf{A}_{13} \\ \mathbf{b}_2^{(2)} &= \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{L}^{-1}\mathbf{b}_1 \end{aligned}$$

Equation (35) must be solved for $\mathbf{x}_t$, and the resulting terms must be inserted into the remaining system matrix, i.e., at all places where $\mathbf{x}_{relax}$ is present.

10. TEARING OR RELAXING?

When dealing with arbitrarily structured sparse matrix systems, it is not always easy to decide, which of the two techniques will work better. First of all, as seen in the multibody example, tearing and relaxing are clearly two distinct methods. *Tearing* reduces always to a (hopefully small) set of non–sparse equations that can afterwards be solved by a full Gaussian elimination, whereas *relaxing* directly generates a Gaussian elimination scheme for sparse matrices. The important point relating to both methods is that sophisticated modelers can anchor the tearing and relaxing information in component model libraries, and hide it from the casual user who would be easily overwhelmed by the demand of placing the *residue* and *relax* operators appropriately in his or her model. Implemented in this fashion, both tearing and relaxing can lead to very efficient simulation code, while preserving the full flexibility of an object–oriented modeling environment, yet without placing unreasonable demands of sophistication on the end user of the modeling tool.

In some cases, both methods are applicable. For example, the drive train problem could as easily have been solved using the *tearing* approach. In that case, the `ShaftS` class would have been augmented in the following way:

```
model class ShaftS
  parameter J
  cut l (w, a / tl)
  cut r (w, a / - tr)

    der(w) = a
    J * a = tl - tr + residue(a)
end
```

Relaxing has a severe drawback in the case of drive trains: components can no longer be connected in an arbitrary fashion. For example, if the gearbox in figure 1 would have been connected in the other way around (just rotating the icon by 180 degrees), the relaxed BLT–transformation would fail. It is easy to check this behavior at equation (12) by relaxing α_1 instead of α_2 in the second row (gear box equation). Since tearing does not share this drawback, and since the algorithm produces the exact same results with the same number of operations as relaxing does (in this example), tearing is clearly the method of choice for drive trains. In this paper, drive trains have only been selected as an introductory example, because the relaxing method can be easily explained by means of such systems.

In principle, tearing could also be used for electrical circuits. The potentials at all nodes would then be selected as tearing variables, whereas Kirchhoff's current–law equations (13) would be selected as residue equations. Unfortunately, this is conceptually difficult, because equations (13) are not defined in model classes. Instead, they are only generated internally at compile time from the connection structure of the circuit, i.e., it is not obvious how to refer to these equations within model classes. Relaxing is better in this case, because the appropriate relaxing structure can be easily defined within model classes. The drawback found for drive trains is not present in electrical circuits. The reason is that there is only one current in each basic electrical component, and therefore, the variable not to be relaxed is unique.

As already shown in the case of multibody systems, both tearing and a combination of relaxing and tearing are possible. The former approach leads to a resulting code with $O(f^3)$ operations. The latter leads to one with $O(f)$ operations. For bigger systems, the latter method is therefore more attractive. Unfortunately, the drawback of the drive trains carries over to multibody systems: when using the **relax** operator, multibody objects must always be connected at opposite cuts (i.e., cut `a` must be connected at cut `b`, and vice–versa), otherwise the relaxed BLT–transformation will fail.

Both methods currently have the disadvantage that they produce slightly less efficient code than a direct implementation of a similar multibody algorithm would. The reason is that the symmetry of matrices present in the equations of multibody systems is not taken into account. For example, tearing of equation (31) does not make use of the symmetry of the matrix. It is not yet clear how symmetry of matrices could be exploited with either tearing or relaxing methods.

On the other hand, this very same feature offers new possibilities. Only a specific class of multibody systems leads to symmetric mass matrices or to symmetric matrices in the relaxing scheme. For example, if Coulomb friction is present in the model, i.e., applied forces are functions of constraint forces, then these matrices are no longer symmetric. This can be seen from equation (31): when Coulomb friction is present, the driving torque in the joint is a linear function of the cut–forces: $\tau = -\hat{\boldsymbol{\mu}}^T \hat{\mathbf{f}}_2$. As a consequence, element 4 in row 5 is changed to $\hat{\mathbf{n}}^T + \hat{\boldsymbol{\mu}}$, and the matrix loses its symmetry. Due to this structural change, multibody programs often have difficulties with handling such systems. However, both the tearing and the relaxing methods will work fine also in the presence of Coulomb friction.

11. CONCLUSIONS

In this paper, a new symbolic sparse matrix technique, called "relaxing," was introduced. This technique, sometimes in conjunction with an alternative approach, called "tearing," is able to exploit special matrix structures, i.e., domain–specific information about a specific type of system, such as a multibody system or an electrical circuit, in the same fashion as the numerical special–purpose solvers of professional packages do. However, and contrary to the domain–specific programs, the relaxing algorithm can be implemented in an object–oriented modeling environment without either compromising the flexibility or restricting the generality of the environment in any way. The *application* of the algorithm to any special kind of system is non–trivial and requires a good deal of physical insight into how the system works, as well as mathematical knowledge of how this type of system is modeled in professional, domain–specific programs. However, once this insight is available, the *implementation* becomes trivial, as seen by the MNA method for electrical circuits and the recursive $O(f)$ algorithms for multibody systems. In the same way as the occasional user of Spice or Saber doesn't have to know anything about the MNA algorithm, the occasional user of Dymola will not be aware of the presence of the **relax** and **residue** operators in the component libraries that he or she is invoking. Only the designer of these libraries needs that knowledge.

References

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

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

[3] W. Armstrong, "Recursive Solution to the Equations of Motion of an n–link Manipulator," *Proc. 5th World Congress on Theory of Machines and Mechanisms*, 1979.

[4] D.S. Bae, and E.J. Haug, "A Recursive Formulation for Constrained Mechanical System Dynamics: Part I. Open Loop Systems," *Mech. Struct. Mach.*, **15**, 1987, pp. 359–382.

[5] H. Brandl, R. Johanni, and M. Otter, "A Very Efficient Algorithm for the Simulation of Robots and Similar Multibody–Systems Without Inversion of the Mass–Matrix," *Proc. IFAC/IFIP Intl. Symp. Theory of Robots*, Vienna/Austria, 1986.

[6] F.E. Cellier, H. Elmqvist, and M. Otter, "Modeling from Physical Principles," *The Control Handbook* (W.S. Levine, ed.), CRC Press, Boca Raton, FL, 1996, pp. 99–108.

[7] I.S. Duff, A.M. Erismann, and J.K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, 1986.

[8] H. Elmqvist, *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, 1978.

[9] H. Elmqvist, *Dymola: Dynamic Modeling Language — User's Guide*, Dynasim AB, Lund, Sweden, 1995.

[10] H. Elmqvist, and M. Otter, "Methods for Tearing Systems of Equations in Object–Oriented Modeling," *Proc. ESM'94, European Simulation Multiconference*, Barcelona, Spain, June 1–3, 1994, pp. 326–332.

[11] H. Elmqvist, M. Otter, and F.E. Cellier, "Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential–Algebraic Equation Systems," Keynote Address, *Proc. ESM'95, European Simulation Multiconference*, Prague, Czech Republic, June 5–8, 1995, pp. xxiii–xxxiv.

[12] R. Featherstone, "The Calculation of Robot Dynamics Using Articulated–Body Inertias," *Intl. J. Robotics Research*, **2**, 1983, pp 13–30.

[13] G.D. Hachtel, R.K. Brayton, and F.G. Gustavson, "The Sparse Tableau Approach to Network Analysis and Design," *IEEE Trans. Circuit Theory*, **CT-18**, 1971, pp. 101–118.

[14] C.W. Ho, A.E. Ruehli, and P.A. Brennan, "The Modified Nodal Approach to Network Analsis," *Proc. Intl. Symp. Circuits & Systems*, San Francisco, 1974, pp. 505–509.

[15] G. Kron, *Diakoptics — The Piecewise Solution of Large–Scale Systems*, MacDonald & Co., London, 1962.

[16] M. Krebs, *Efficient Simulation of Power–Electronic Switching Circuits Using Inline Integration*, MS Thesis, Dept. of Electr. & Comp. Engr., University of Arizona, 1996.

[17] R.S.H. Mah, *Chemical Process Structures and Information Flows*, Butterworths, 1990.

[18] H.A. Mantooth, and M. Vlach, "Beyond SPICE with Saber and MAST," *Proc. IEEE Intl. Symp. Circuits and Systems* San Diego, CA, 1993, pp. 77–80.

[19] W.J. McCalla, *Fundamentals of Computer–Aided Circuit Simulation*, Kluwer Academic Publishers, 1988.

[20] L.W. Nagel, *SPICE2: A computer program to simulate semiconductor circuits*, Electronic Research Laboratory, University of California, Berkeley, ERL–M 520, 1975.

[21] M. Otter, and F.E. Cellier, "Software for Modeling and Simulating Control Systems," *The Control Handbook* (W.S. Levine, ed.), CRC Press, Boca Raton, FL, 1995, pp. 415–428.

[22] W. Rulka, "SIMPACK — A Computer Program for Simulation of Large–motion Multibody Systems," *Multibody Systems Handbook* (W. Schiehlen, ed.), Springer–Verlag, 1990.

[23] R.C: Smith, and E.J. Haug, "DADS — Dynamic Analysis and Design System," *Multibody Systems Handbook* (W. Schiehlen, ed.), Springer–Verlag, 1990.