INLINE INTEGRATION: A NEW MIXED SYMBOLIC/NUMERIC APPROACH FOR SOLVING DIFFERENTIAL–ALGEBRAIC EQUATION SYSTEMS

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

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

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 solving differential–algebraic equation systems using a mixed symbolic and numeric approach. Discretization formulae representing the numerical integration algorithm are symbolically inserted into the differential–algebraic equation model. The symbolic formulae manipulation algorithm of the model translator treats these additional equations in the same way as it treats the physical equations of the model itself, i.e., it looks at the augmented set of algebraically coupled equations and generates optimized code to be used with the underlying simulation run–time system. For implicit integration methods, a large nonlinear system of equations needs to be solved at every time step. It is shown that the presented uniform treatment of model equations and discretization formulae often leads to a significant reduction of the number of iteration variables and therefore to a substantial increase in execution speed.

In a large mechatronics system consisting of a six degree–of–freedom robot together with its motors, drive trains, and control systems, this approach led to a speedup factor of more than ten.

Keywords: Inline integration; tearing structure; symbolic formulae manipulation; differential–algebraic equation solving; simulation efficiency.

INTRODUCTION

There is a conviction of large segments of the simulation community that it is important to separate the knowledge about the numerical integration algorithms to be used during the execution of a continuous–time simulation program from knowledge relating to either the physical system to be studied or the experiment to be performed on that system. The major reason for this perceived need is that, in this way, the details of the underlying numerical algorithms can be hidden from the simulation user. The average simulationist should not be bothered to have to think about the numerics of the solution approach.

The simulation software designers do their best to make the simulation users forget that discrete mathematics are involved at all in the numerical solution of their continuous–time simulation problems. Differentiation or integration operators are offered in the modeling language that make the user believe that the simulation program knows how to solve differential equations. In reality, it is the task of either the modeling or the simulation software to convert the continuous–time problem to an –in some way equivalent– discrete–time problem, usually with variable time increments, that can then be solved through iteration. In traditional approaches, this responsibility is assigned to the simulation software. In the here presented new approach, it rests with the modeling software. In either case, it is important to protect the average simulation user from having to be aware of this conversion.

As the demand for models of systems of ever increasing complexity grew, so did the need for organizing and encapsulating knowledge about these systems. It was no longer sufficient to separate the knowledge about the model from the integration method and the experiment description. The knowledge about the model itself needed to be organized. This led to the design of object–oriented modeling software, such as Dymola (Elmqvist, 1978; Cellier and Elmqvist, 1993; Elmqvist, 1995) and Omola/Omsim (Andersson, 1994). These languages allow the user to specify the physical laws that govern the behavior of a physical entity in terms of declarative equations. An interface description declares, which properties (variables) of the system are shared with other systems (but without defining the direction of information flow), and which others are hidden from the outside. Models of such entities can be plugged together in a fashion resembling the assembly of physical plants from their component systems.

It is, however, important to understand what drove the design of object–oriented modeling software.

Object–orientation here supports the *human user* of the software in organizing his or her knowledge about the system under study. It is not the simulation run–time software that is supported by this segmentation of knowledge. On the contrary, if the segmentation of knowledge into parts related to physical subsystems were preserved down to the level of the run–time execution of the simulation program, the execution efficiency would be terrible. Thus, the first step in the compilation of an object–oriented model consists in collecting all the equations from the individual submodels and from the additional equations that describe the couplings between submodels into an amorphous heap of equations, throwing all structuring information away. This accumulated heap of equations is analyzed to find an execution sequence that will compute one value for each of the variables involved, ensuring that the sizes of the remaining systems of tightly coupled algebraic equation systems are minimized. This process is called the *partitioning* of equations or BLT-transformation.

It turns out that any artificial constraint imposed when seeking the optimal solution to the partitioning problem is potentially harmful to the optimization of the run–time code. In this paper, it will be shown that even the last of the barriers, the separation between the model and the numerical algorithms must come down in order to enable a symbolic translator, such as Dymola, to generate more efficient simulation run–time code. In the case of a large mechatronics system consisting of a six degree–of–freedom robot together with its motors, drive trains, and control system, this led to a speedup factor of more than ten.

Designers of domain–specific simulation software have recognized long ago that they could improve the execution efficiency of their simulation programs by providing the numerical solver with structural information about the model to be simulated. In SPICE (Nagel, 1975), the most successful among the analog electrical and electronic circuit programs, the so–called transient analysis (simulation) is performed by treating the node voltages as iteration variables in an implicit numerical solution scheme. With the node voltages assumed known, the charges stored in the capacitors can be computed by nonlinear static functions. From there, the branch currents through the capacitors are computed by replacing the differentiation operator by a discrete approximation formula. This is done for each capacitor separately. Parasitic capacitances inside the transistor models are approximated differently, since they are known to be small. Several proposals have recently been formulated to exploit the special structure of multibody systems within the numerical solver (Andrzejewski *et al.*, 1993; Cardona and Gérardin, 1993; Lubich *et al.*, 1993) in order to improve the execution efficiency of special–purpose multibody system simulators.

Unfortunately, these approaches are of limited use outside their intended application area, because only specific types of systems can be handled by them, and because the user interface becomes quickly rather complicated in order to provide the numerical integrator with the necessary information about the specific model structure. What has been lacking so far is a powerful symbolic formulae manipulation tool that can analyze the resulting equation structure and automatically generate the appropriate structuring information for the underlying numerical system solver.

The methodology presented in this paper discusses precisely such an approach. It is very general in scope, and can be implemented with reasonable effort as part of an object–oriented modeling tool. However, the numerical solver must be modified for this approach to work. It is not feasible to translate a general model description into a form suitable for an off–the–shelf numerical system solver without sacrificing execution efficiency. Instead, it is important that the equations describing the numerical discretization of the integral or differential operators be merged with the model equations symbolically prior to analyzing the structure of the resulting equation system.

THE BASIC IDEA OF INLINE INTEGRATION

When SCS, in 1967, launched a commendable effort to standardize the Continuous System Simulation Languages (CSSLs) (Augustin *et al.*, 1967), they adopted the world view that continuous–time systems can essentially be expressed as *state–space models*, represented through a set of *Ordinary Differential Equations (ODEs)*:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \quad ; \quad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{1}$$

where $\mathbf{x}$ is the vector of state variables, t denotes time, and $\mathbf{f}$ is a set of assignment statements specifying how the derivatives are computed assuming that the state variables are known. The computational causality thus needs to be specified in the model. The CSSL user interface provided for a more convenient way of specifying state–space models, but it was clearly designed as a frontend to numerical subroutines for solving non–stiff ODE's.

Solving (1) by any explicit integration method is straightforward. In the most simple case, using the forward Euler method, the derivative of the state vector is approximated by a forward difference formula:

$$\dot{\mathbf{x}}(t_n) = \dot{\mathbf{x}}_n \approx \frac{\mathbf{x}_{n+1} - \mathbf{x}_n}{h} \tag{2}$$

where $\mathbf{x}_{n+1} = \mathbf{x}(t_{n+1})$ is the unknown value of $\mathbf{x}$ at the new time instant $t_{n+1} = t_n + h$, $\mathbf{x}_n = \mathbf{x}(t_n)$ is the

known value of $\mathbf{x}$ at the previous time instant t_n, and h is the chosen step size. Inserting (2) into (1) leads to the following recursion formula:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}(\mathbf{x}_n, t_n) \quad ; \quad \mathbf{x}_0 \text{ is known} \tag{3}$$

which is used to "solve" the ODE.

Unfortunately, explicit integration methods are no longer well suited if systems are stiff or contain algebraic loops. In such cases, implicit integration methods are much more appropriate. In the simplest case, using the backward Euler method, the derivative of the state vector is approximated by a backward difference formula, leading to:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}(\mathbf{x}_{n+1}, t_{n+1}) \tag{4}$$

which must be solved for $\mathbf{x}_{n+1}$, given $\mathbf{x}_n$ and t_{n+1}. (4) is a *nonlinear* equation for $\mathbf{x}_{n+1}$, which has to be solved in every time step. It can be shown that Newton iteration has to be used to solve this equation in order to maintain proper convergence properties, cf. e.g. (Cellier, 1995); fixed–point iteration is not useful if the system is stiff.

For a large class of implicit integration algorithms the structure of the discretization equations are the same, in particular:

$$\mathbf{x} = h \cdot \dot{\mathbf{x}} + old(\mathbf{x}) \tag{5}$$

For notational convenience, the unknown values $\mathbf{x}_{n+1}$ and $\dot{\mathbf{x}}_{n+1}$ have been abbreviated by $\mathbf{x}$ and $\dot{\mathbf{x}}$, respectively. The known scalar h depends on the step size and on method–specific constants, whereas $old(\mathbf{x})$ is a function of known values of $\mathbf{x}$ at previous time instants. Especially, the *Backward Difference Formulae (BDF)* (Gear, 1971) of any order, which are the most widely used formulae for numerically solving stiff systems, fall into this category. For example, the third order BDF can be written as ($\bar{h}$ is the step size):

$$\mathbf{x}_{n+1} = \tfrac{6}{11}\bar{h} \cdot \dot{\mathbf{x}}_{n+1} + \left(\tfrac{18}{11}\mathbf{x}_n - \tfrac{9}{11}\mathbf{x}_{n-1} + \tfrac{2}{11}\mathbf{x}_{n-2}\right)$$

which has clearly an equation structure according to (5). Inserting the general discretization scheme (5) into (1) again leads to the same nonlinear equation as in the backward Euler case, with the only exception that the known variables now have a different interpretation:

$$\mathbf{x} = old(\mathbf{x}) + h \cdot \mathbf{f}(\mathbf{x}, t) \tag{6}$$

For a general function $\mathbf{f}(\mathbf{x}, t)$, there is no way to avoid the (expensive) Newton iteration scheme to solve (6) for $\mathbf{x}$ in any time step. However for specific models, the situation is different. If, for example, two linear filters are connected in series, as shown in figure 1, the

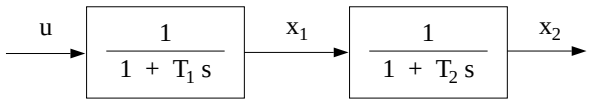

Figure 1: Two Filters in Series

overall system is described by the following equations:

$$\begin{aligned} T_1\dot{x}_1 + x_1 &= u(t) \\ T_2\dot{x}_2 + x_2 &= x_1 \end{aligned}$$

Utilizing the discretization formula (5) leads to an explicitly solvable sequence of equations to compute the four unknown variables at the new time instant:

$$\begin{aligned} \dot{x}_1 &:= (u - old(x_1))/(T_1 + h) \\ x_1 &:= h\dot{x}_1 + old(x_1) \\ \dot{x}_2 &:= (x_1 - old(x_2))/(T_2 + h) \\ x_2 &:= h\dot{x}_2 + old(x_2) \end{aligned}$$

As can be seen, no Newton iteration is needed for this special type of system, because the linear equations can be solved symbolically. Let us analyze yet another type of system where higher derivatives appear that are transformed to state–space form in the standard way:

$$\dot{\mathbf{x}}_1 = \mathbf{x}_2 \tag{7}$$

$$\dot{\mathbf{x}}_2 = \mathbf{f}(\mathbf{x}_1, \mathbf{x}_2, t) \tag{8}$$

Without knowing the structure of this system, a Newton iteration about $2n$ equations is necessary, where n is the dimension of function $\mathbf{f}$. However, inserting the discretization formula (5) leads to:

$$\mathbf{x}_2 = h \cdot \mathbf{f}(h\mathbf{x}_2 + old(\mathbf{x}_1), \mathbf{x}_2, t) + old(\mathbf{x}_2) \tag{9}$$

$$\mathbf{x}_1 := h\mathbf{x}_2 + old(\mathbf{x}_1) \tag{10}$$

i.e., to a nonlinear system of n equations to determine $\mathbf{x}_2$, and an explicitly solvable set of n equations to compute $\mathbf{x}_1$.

To summarize, for the solution of non–stiff ODEs with explicit integration methods, the traditional function interface (1) is well suited. It is not necessary to know the structure of the right hand side equations, $\mathbf{f}$, inside the integrator, because this information will not help in speeding up the simulation. The situation is completely different for ODEs that are stiff or contain algebraic loops. Implicit integration methods lead to nonlinear systems of equations. If the structure of the model equations and the discretization formula are known, the efficiency of the simulation can often be enhanced, as shown by examples. The function interface (1) is not helpful in such a situation, since the structure of the equations, such as filters in series or higher

derivatives, is not reported to the nonlinear equation solver inside the integrator.

It is now easy to explain how some domain–specific packages, as mentioned in the introduction, enhance the efficiency of the simulation. Information is added to the function interface in order to report some supported equation structure, such as the presence of higher–order derivative equations, to the integrator. Usually these packages use this information to solve the nonlinear system of the discretized equations more efficiently.

This concept can be generalized (Elmqvist, 1993). The generalization shall be denoted as *inline integration* in the sequel. Inline integration requires a modified integrator interface. The integrator maintains information about the discretization of the state variables, such as the known quantities h and $old(\mathbf{x})$, and provides initial estimates for the values of $\mathbf{x}$ and $\dot{\mathbf{x}}$ at the current time instant. The modeling software sets up the nonlinear system of discretized equations, and solves it, utilizing the known structure of the equations, by calling upon a run–time library function for Newton iteration on a minimal set of algebraically coupled nonlinear equations. The function returns the actual values of $\mathbf{x}$ and $\dot{\mathbf{x}}$ at the new time instant, or complains that the Newton iteration did not converge within a specific number of iterations defined by the integrator. It should be noted that error estimation, step–size and order control, details of the discretization formula used (i.e., the computation of h from the step size, and the evalutaion of $old(\mathbf{x})$ from the known previous values of $\mathbf{x}$) are still in the domain of the numerical solver. Only the discretization of the state equations and the solution of the (usually nonlinear) system of discretized equations have been moved into the domain of the model. This modified interface includes all domain–specific approaches that were previously in use as special cases.

The generation of the new model interface should be made *automatic*, i.e., by a program. It will be shown, how this can be done starting from an object–oriented, high–level description of a model. There are several subtle issues to be considered in order that such a translation process leads to robust code. Especially, it must be guaranteed that, in the worst case, i.e., when no structural information can be utilized, the generated code has exactly the same numerical properties as if the traditional standard interface would have been used. In all other cases, it should perform better.

In order to be able to explain the details of the automatic generation of inline discretized model code, several prerequisites are necessary that shall be discussed in the following sections: the discretization of general continuous–time dynamical models, the block lower–triangular (BLT) transformation of sets of algebraic equations, and the tearing method to formalize the translation process of algebraically coupled equation systems exploiting structural properties of these systems to enhance the execution efficiency in their solution.

THE DISCRETIZATION OF DAES

Physical systems, such as electrical circuits, mechanical systems, or chemical plants, often lead naturally to models described by sets of *differential–algebraic equations (DAEs)* of the following general form:

$$\mathbf{0} = \mathbf{f}\left(\dot{\mathbf{y}}, \mathbf{y}, t\right) \quad ; \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad \dot{\mathbf{y}}(t_0) = \dot{\mathbf{y}}_0 \tag{11}$$

where $\mathbf{y}(t)$ is a vector of unknown variables that may also appear in the equations in differentiated form.

The most widely used general–purpose DAE code, DASSL (Petzold, 1983), solves (11) by using a BDF–discretization, cf. (Brenan *et al.*, 1989) for implementational details. In particular, $\dot{\mathbf{y}}$ is approximated by a backward differentation formula, leading to the following discretized equations:

$$\mathbf{0} \;=\; \mathbf{f}\left(\frac{\mathbf{y} - old(\mathbf{y})}{h}, \mathbf{y}, t\right) \tag{12}$$

that must be solved for $\mathbf{y}$. The standard Newton iteration scheme used in DASSL to solve (12) is given by the following equations:

$$\left(\frac{1}{h}\mathbf{J}_{\dot{y}} + \mathbf{J}_y\right) \cdot \boldsymbol{\delta}^l \;=\; -\mathbf{f}^l\left(\dot{\mathbf{y}}^l, \mathbf{y}^l, t\right) \tag{13}$$

$$\mathbf{y}^{l+1} \;=\; \mathbf{y}^l + \boldsymbol{\delta}^l \tag{14}$$

$$\dot{\mathbf{y}}^{l+1} \;=\; \dot{\mathbf{y}}^l + \frac{1}{h}\boldsymbol{\delta}^l \tag{15}$$

where index l denotes the previous (known) iterate and index $l+1$ denotes the unknown current iterate. The Jacobians are evaluated at one of the previous time steps, and are held constant for as long as it is possible. They are defined by:

$$\mathbf{J}_{\dot{y}} = \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{y}}} \quad ; \quad \mathbf{J}_y = \frac{\partial \mathbf{f}}{\partial \mathbf{y}} \tag{16}$$

On a first glance, one may think that this type of discretization scheme should also be used in inline integration. However, such an approach would lead to unnecessary difficulties:

1. Both when calculating the Newton iteration matrix as well as when updating $\dot{\mathbf{y}}^{l+1}$, division by h takes place. The scalar h depends linearly on the step size, thus $h \to 0$ as the step size approaches zero. It must therefore be expected that difficulties will occur when the step size becomes very small, since, in the limit, a division by zero takes place.

2. The problem is even more serious than initially expected, as can be seen when multiplying (13) by h:

$$(\mathbf{J}_{\dot{y}} + h \cdot \mathbf{J}_y) \cdot \boldsymbol{\delta}^l \;=\; -h \cdot \mathbf{f}^l\,(\dot{\mathbf{y}}^l, \mathbf{y}^l, t)$$

For $h = 0$, the iteration matrix reduces to $\mathbf{J}_{\dot{y}}$. Unfortunately, this matrix is singular whenever algebraic equations are present in the equation set, i.e., when some elements of $\mathbf{y}$ do not appear in differentiated form in the equation set. In this case, the Newton iteration will no longer work. Consequently, the Newton iteration matrix in (13) becomes ill–conditioned for small step sizes, provided purely algebraic equations appear in (11).

3. In order to use DASSL, consistent initial conditions $\mathbf{y}_0$, $\dot{\mathbf{y}}_0$ must be provided, such that the DAE is satisfied at initial time. In general, it is difficult to provide such initial conditions. A simulation program should support the user in this respect. If the DAE is given in the form of (11), this is not easy.

All the aforementioned difficulties disappear, provided some small changes are made. First, it should be explicitly noted whether a variable does or does not appear in differentiated form in the model. Due to this requirement, the DAE is specified in the following form:

$$\mathbf{0} = \mathbf{f}\,(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t) \quad ; \quad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{17}$$

where $\mathbf{x}$ is the vector of unknown variables that appear in the model in differentiated form, whereas $\mathbf{w}$ is the vector of unknown purely algebraic variables. Note, that $dim(\mathbf{f}) = dim(\mathbf{x}) + dim(\mathbf{w})$, and that $\mathbf{y} = [\mathbf{x}; \mathbf{w}]$. Second, the discretization procedure should replace $\mathbf{x}$ as a function of $\dot{\mathbf{x}}$ and not the other way around, as done in DASSL. If $\mathbf{x}$ is replaced in (17), using the general discretization formula (5), one obtains:

$$\mathbf{0} = \mathbf{f}\,(\dot{\mathbf{x}}, h\dot{\mathbf{x}} + old(\mathbf{x}), \mathbf{w}, t) \tag{18}$$

Applying standard Newton iteration to (18) leads to the following iteration procedure:

$$[\mathbf{J}_{\dot{x}} + h\mathbf{J}_x, \mathbf{J}_w] \begin{bmatrix} \boldsymbol{\delta}^l_{\dot{x}} \\ \boldsymbol{\delta}^l_w \end{bmatrix} \;=\; -\mathbf{f}^l\,(\dot{\mathbf{x}}^l, \mathbf{x}^l, \mathbf{w}^l, t) \tag{19}$$

$$\dot{\mathbf{x}}^{l+1} \;=\; \dot{\mathbf{x}}^l + \boldsymbol{\delta}^l_{\dot{x}} \tag{20}$$

$$\mathbf{w}^{l+1} \;=\; \mathbf{w}^l + \boldsymbol{\delta}^l_w \tag{21}$$

$$\mathbf{x}^{l+1} \;=\; \mathbf{x}^l + h \cdot \boldsymbol{\delta}^l_{\dot{x}} \tag{22}$$

Obviously, a vanishing step size will not lead to a division by zero. If $h = 0$, the Newton iteration matrix reduces to:

$$[\mathbf{J}_{\dot{x}}\ ,\ \mathbf{J}_w]$$

It can be easily proven that this matrix is non–singular, provided the DAE (17) has perturbation index 1 (a necessary and sufficient condition), cf. e.g. (Otter, 1995). In other words, if the purely algebraic equations in the DAE are not "nasty," the iteration matrix is non–singular in the limit $h = 0$.

Finally, (18) has the practical advantage that the discretized DAE reduces to the original DAE (17) when h is set to zero. This property can be exploited for the calculation of consistent initial conditions. The user has to provide initial conditions $\mathbf{x}_0$ and first guesses for $\dot{\mathbf{x}}_0$, and $\mathbf{w}_0$. Before the integration starts, the (Chord–) Newton iteration is replaced by a more robust (yet more expensive) Newton–Raphson iteration, and $h = 0$, $old(\mathbf{x}) = \mathbf{x}_0$ is set as indicated. As a result, the discretized DAE (18) reduces to a nonlinear equation in $\dot{\mathbf{x}}_0$ and $\mathbf{w}_0$, which is solved by Newton–Raphson iteration.

To summarize, in a fully–implicit DAE, the discretization procedure has to replace $\mathbf{x}$ by a function of $\dot{\mathbf{x}}$ in accordance with (5). Only in the ODE case, the alternative of replacing $\dot{\mathbf{x}}$ by a function of $\mathbf{x}$, is meaningful (cf. (6)).

BLT–TRANSFORMATION

In order to be able to discuss the technique to transform a general object–oriented model automatically down to a suitably discretized system, 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, nonlinear system of equations that has to be solved for the unknown variables $\mathbf{z}$:

$$\mathbf{0} \;=\; \mathbf{h}(\mathbf{z}) \tag{23}$$

By permutation of equations and variables, it is possible to transform this system of equations to a 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:

$$\begin{aligned} h_1\,(z_1, z_3) &= 0 \\ h_2\,(z_2) &= 0 \\ h_3\,(z_1, z_2) &= 0 \end{aligned} \qquad \mathbf{S}_1 = \begin{array}{c} \begin{array}{ccc} z_1 & z_2 & z_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \end{array}$$

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 kth variable (k^{th} column) occurs in the ith equation (i^{th} row), or not. By permuting equations and variables, this set of equations can be brought to BLT–form:

$$\begin{aligned} h_2\,(z_2) &= 0 \\ h_3\,(z_1, z_2) &= 0 \\ h_1\,(z_1, z_3) &= 0 \end{aligned} \qquad \mathbf{S}_2 = \begin{array}{c} \begin{array}{ccc} z_2 & z_1 & z_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \end{array}$$

This process is also called the *partitioning* of the set of equations. The strictly lower triangular form of the permuted structure incidence matrix characterices the fact that the nonlinear equations can be solved one at a time in a given sequence. We start by solving h_2 for z_2, then we can solve h_3 for z_1, and finally, we can determine z_3 from h_1. If the variable to be solved for appears linearly in an equation, that equation can be put into explicit form by simple formula 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., a quasi lower–triangular form in which blocks of dimension ≥ 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 e.g. be found in (Duff *et al.*, 1986).

TEARING

Tearing, introduced by (Kron, 1962), is a simple technique to reduce large systems of linear or nonlinear algebraic equations to smaller systems of equations. This technique has, for example, been applied successfully for static calculations in chemical engineering (Boston, 1980; Simandl and Svrcek, 1991). Tearing is used in the sequel to formalize the automatic generation of discretized model equations and to enhance the run–time efficiency of BLT transformed equations even further.

Consider a set of nonlinear algebraically coupled equations $\mathbf{h}$ to be solved for the unknown vector $\mathbf{z}$:

$$\mathbf{0} = \mathbf{h}(\mathbf{z}) \tag{24}$$

Tearing means breaking algebraic loops in the dependency structure of equations and variables. A subset of the vector $\mathbf{z}$, called $\mathbf{z}_1$, is chosen as *tearing variables*. A subset of $\mathbf{h}$, called $\mathbf{h}_1$, are chosen as *residual equations*. The choices are made in such a way that the remainder of $\mathbf{z}$, called $\mathbf{z}_2$, can be calculated in sequence utilizing the remaining equations, $\mathbf{h}_2$, under the assumption that the $\mathbf{z}_1$ variables are known, i.e.:

$$\mathbf{z}_2 = \mathbf{h}_2(\mathbf{z}_1) \tag{25}$$

$$\mathbf{0} = \mathbf{h}_1(\mathbf{z}_1, \mathbf{z}_2) \tag{26}$$

This system of equations can be solved by Newton iteration over the tearing variables $\mathbf{z}_1$. The solver provides a new guess for $\mathbf{z}_1$. With (25), the corresponding variables $\mathbf{z}_2$ are calculated. Finally, the residual (26) is computed and returned to the solver. As can be seen, tearing reduces the dimension of the iterated system of equations from $dim(\mathbf{h}_1) + dim(\mathbf{h}_2)$ down to $dim(\mathbf{h}_1)$.

The optimal selection of tearing variables and residual equations is not a trivial task. No efficient algorithms are currently known to automate it. Exhaustive search algorithms to determine an optimal tearing structure are unfortunately of exponential complexity. However, in (Elmqvist and Otter, 1994), it is shown that physical insight may suggest appropriate tearing variables and residual equations. Often this information can be stored in class libraries of an object–oriented modeling language, such that the tearing structure is totally hidden from the user.

In (Elmqvist and Otter, 1994), the selected tearing variables and residual equations are uniquely specified by an operator "$residue(z_i)$" that has to be added to the desired model equation j, e.g. in an appropriate class library. This operator indicates to the modeling software that variable z_i shall be used as tearing variable, and that equation j shall be used as the corresponding residual equation. For example, tearing in (25,26) is uniquely characterized by:

$$\mathbf{z}_2 = \mathbf{h}_2(\mathbf{z}_1) \tag{27}$$

$$residue(\mathbf{z}_1) = \mathbf{h}_1(\mathbf{z}_1, \mathbf{z}_2) \tag{28}$$

The tearing technique allows the automated transformation of model equations to their discretized form in a simple way. Let us assume for now that the equations of a model are specified in ODE form (1). In order to arrive at a set of discretized equations (6), one could proceed as shown earlier in the hand calculation, i.e., $\dot{\mathbf{x}}$ is replaced by $(\mathbf{x} - old(\mathbf{x}))/h$, and afterwards all equations are symbolically multiplied by h. However, such an approach cannot easily be generalized to models specified in DAE form. Tearing provides for a much more elegant formulation:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{29}$$

$$\mathbf{x} = old(\mathbf{x}) + h \cdot \dot{\mathbf{x}} + residue(\mathbf{x}) \tag{30}$$

Thereby, the model equations are kept unchanged and the additional discretization equations (30) are simply added. The Newton solver will provide an estimate for $\mathbf{x}$, then $\dot{\mathbf{x}}$ is determined from the state equation (29), and finally, the residual of the nonlinear equation is computed via (30) and is returned to the solver. As a result, the tearing technique ends up with exactly the same equations that one would derive from hand calculation (6).

The tearing technique is also well suited for enhancing the efficiency of the solution of a nonlinear system of discretized equations. Let us discuss, for example, a nonlinear control system with linear feedback, as shown in figure 2. This rather typical control system

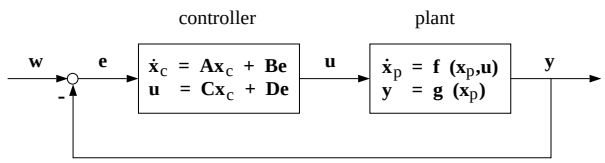

Figure 2: Control System With Feedback

can be described by the following equations:

$$\begin{aligned} \mathbf{e} &= \mathbf{w} - \mathbf{y} \\ \dot{\mathbf{x}}_c &= \mathbf{A}\mathbf{x}_c + \mathbf{B}\mathbf{e} \\ \mathbf{u} &= \mathbf{C}\mathbf{x}_c + \mathbf{D}\mathbf{e} \\ \dot{\mathbf{x}}_p &= \mathbf{f}(\mathbf{x}_p, \mathbf{u}) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}_p) \end{aligned}$$

Using the standard discretization technique described above leads to:

$$\begin{aligned} \mathbf{y} &= \mathbf{g}(\mathbf{x}_p) \\ \mathbf{e} &= \mathbf{w} - \mathbf{y} \\ \mathbf{u} &= \mathbf{C}\mathbf{x}_c + \mathbf{D}\mathbf{e} \\ \mathbf{x}_c &= old(\mathbf{x}_c) + h \cdot (\mathbf{A}\mathbf{x}_c + \mathbf{B}\mathbf{e}) + residue(\mathbf{x}_c) \\ \mathbf{x}_p &= old(\mathbf{x}_p) + h \cdot \mathbf{f}(\mathbf{x}_p, \mathbf{u}) + residue(\mathbf{x}_p) \end{aligned}$$

That is, given the two sets of tearing variables, $\mathbf{x}_c$ and $\mathbf{x}_p$, all other variables can be calculated, especially the set of $dim(\mathbf{x}_c) + dim(\mathbf{x}_p)$ residual equations. As a result, the same Newton iteration is obtained as if the discretization would have been done in the integrator.

As can be seen, the residue equations of the controller are linear in the unknown controller states. It is therefore possible to solve these equations directly, and to remove the controller states from the iteration variables:

$$\begin{aligned} \mathbf{y} &= \mathbf{g}(\mathbf{x}_p) \\ \mathbf{e} &= \mathbf{w} - \mathbf{y} \\ \mathbf{x}_c &= (\mathbf{I} - h\mathbf{A})^{-1}(old(\mathbf{x}_c) + h\mathbf{B}\mathbf{e}) \\ \mathbf{u} &= \mathbf{C}\mathbf{x}_c + \mathbf{D}\mathbf{e} \\ \mathbf{x}_p &= old(\mathbf{x}_p) + h \cdot \mathbf{f}(\mathbf{x}_p, \mathbf{u}) + residue(\mathbf{x}_p) \end{aligned}$$

That is, Newton iteration is applied to the much smaller set of equations of dimension $dim(\mathbf{x}_p)$. This is advantageous, if the explicit calculation of $\mathbf{x}_c$ is cheap. There is a lot of freedom in the implementation of a controller law. Usually, specially structured matrices are used, such as the controller–canonical form with a sparse $\mathbf{A}$ matrix and a cheaply computable inverse. Such structures often occur in a natural way when simple low order blocks are connected together to build up the controller. If the Jordan–canonical form is used, the inversion becomes trivial.

Evidently, the above example is quite generic. However, it illustrates well how the set of Newton iteration variables in a model can be reduced by exploiting knowledge about the model structure, here the linearity of a submodel. It is important to keep the number of Newton iteration variables as small as possible, since this reduces the sizes of the Jacobians, and since it will improve the convergence speed of the iteration.

INLINE INTEGRATION OF DAES

We have now all pieces together to discuss the general algorithm to transform a DAE down to a suitable discretized form in an automated manner.

Inline integration of a DAE:

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t) \quad ; \quad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{31}$$

is done in the following five steps.

(1) The system is transformed to BLT–form, assuming that $\mathbf{x}$ is known, and that $\mathbf{w}$ and $\dot{\mathbf{x}}$ are unknown. As a result, the systems of equations of minimal dimensions are determined that cannot be solved explicitly.

(2) For every x_i with the property that $\dot{x}_i$ can be explicitly solved for in the partitioned equations (i.e., the corresponding diagonal block of the structure incidence matrix is of dimension one), add the equation:

$$x_i = h \cdot \dot{x}_i + old(x_i) + residue(x_i) \tag{32}$$

For all other x_j, add the same equation but without the term $residue(x_j)$.

(3) If the assigned equation of $\dot{x}_j$ or w_k appears in an algebraic loop (a diagonal block of dimension larger than one), add the term "$residue(\dot{x}_j)$" or "$residue(w_k)$" to the corresponding model equation.

(4) If a declaration not to tear $x_i, \dot{x}_j, w_k$ or terms of the form "$residue(x_i)$", "$residue(\dot{x}_j)$" or "$residue(w_k)$" are already present in the model equations, remove the corresponding residue operators which have been added in (2) or (3). This rule is needed in order to be able to select specific tearing structures in the model class libraries. This feature enables the designer of such libraries to override the default tearing mechanism and select tearing structures that are better suited for the application at hand.

(5) Transform the augmented system of equations to BLT–form, assuming that $\mathbf{w}$, $\dot{\mathbf{x}}$ *and* $\mathbf{x}$ are unknown,

thereby utilizing the tearing information. As a result, the nonlinear systems of equations of the discretized model equations are produced.

Let us discuss some special cases of this algorithm.

If a DAE consists of a single large algebraic loop, in which *all* state derivatives $\dot{\mathbf{x}}$ and algebraic variables $\mathbf{w}$ are involved, the above algorithm will generate the following equations due to steps (2) and (3):

$$\begin{aligned} \mathbf{x} &= h \cdot \dot{\mathbf{x}} + old(\mathbf{x}) \\ \mathbf{0} &= \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t) + residue(\begin{bmatrix} \dot{\mathbf{x}} \\ \mathbf{w} \end{bmatrix}) \end{aligned}$$

i.e., the already discussed basic discretized form of general DAEs (18), where $\dot{\mathbf{x}}$ and $\mathbf{w}$ are used as Newton iteration variables.

If a DAE can be transformed to strictly lower–triangular form, i.e. ODE–form, then no algebraic loops are present, and the following equations will be generated again due to steps (2) and (3):

$$\begin{aligned} \mathbf{x} &= h \cdot \dot{\mathbf{x}} + old(\mathbf{x}) + residue(\mathbf{x}) \\ \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, t) \end{aligned}$$

i.e., the already discussed basic discretized form of ODEs (6), where the state variables, $\mathbf{x}$, are used as iteration variables.

To summarize, in the worst case, the inline integration algorithm will produce the same discretization form as if the discretization procedure would be done directly in the integrator using the standard interfaces. However, in most practical cases, the algorithm will perform considerably better, i.e., either the dimension of the algebraic nonlinear system to be iterated is smaller, or the overall system is broken down into several smaller systems that can be iterated separately. Note that, as shown in the controller example presented in the section entitled "Tearing", it often makes sense to eliminate linear dynamic subsystems from the discretized equations, such that the reduced Newton iteration contains only variables from nonlinear equations.

A MORE DETAILED EXAMPLE

Let us consider the following nonlinear plant model and controller with complex poles.

It can be described by the following equations:

$$\begin{aligned} \dot{x} &= f(x, u, t) \\ y &= g(x) \\ \dot{x}_1 &= x_2 \end{aligned}$$

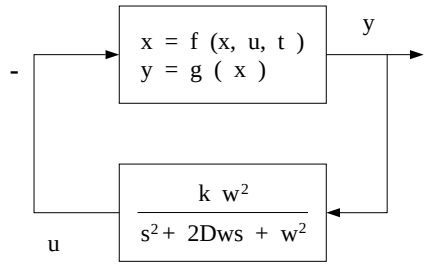

Figure 3: Nonlinear Plant With Feedback Loop

$$\begin{aligned} \dot{x}_2 &= -2\omega D x_2 - w^2 x_1 + w^2 y \\ u &= -k x_1 \end{aligned}$$

This model is completely harmless from an ODE perspective. It presents itself already in state–space form, and any explicit integration algorithms can solve this problem easily, as long as the equations are not stiff.

Use of a numerical DAE solver would require the user to specify for the solver, which are the state variables, and to formulate the corresponding state equations in residual form. In the above example, it would then apply Newton iteration to the resulting three–variable system, in order to keep the residuals close to zero at all times.

This model is a special case of the general control loop discussed in the section entitled "Tearing." Consequently, it is easy to automatically discretize the model using the earlier explained technique. Thereby, the linear part of the discretized equations can be solved explicitly, so that only the nonlinear plant equation has to be solved by Newton iteration.

Using the object–oriented modeling language Dymola, the equations of the system can be directly programmed. Optionally, Dymola transforms the equations down to discretized equations, as explained in the last section, and outputs the result in the form of C–code:

```
/* SORTED AND SOLVED EQUATIONS */

/* SYSTEM OF 7 SIMULTANEOUS NONLINEAR EQUATIONS */
/* EQUATIONS */
/*    derx + residuederx = f(x,u,Time); */
/*    x = h*derx + oldx; */
/*    y = g(x); */
/*    derx2 + linresderx2 + 2*w*D*x2 + w*w*x1 */
/*       = w*w*y */
/*    x2 = h*derx2 + oldx2; */
/*    x1 = h*x2 + oldx1; */
/*    u = k*x1; */

/* SOLVING NONLINEAR SYSTEM OF EQUATIONS */
    QSol[0] = derx;

    QNLnr = 1;
    QNnl = 1;
```

```
    QiOpt = 2;
    QInfRev  = -1;

Iter1 :
    if (QInfRev > 0) {

/* NONLINEAR TEARING VARIABLES AND RESIDUES */
/*         derx          residuederx */

/* TORN NONLINEAR EQUATIONS */
      x = h*derx + oldx;
      y = g(x);

/* SYSTEM OF 3 SIMULTANEOUS LINEAR EQUATIONS */
/* EQUATIONS */
/*    derx2 + linresderx2 + 2*w*D*x2 + w*w*x1 */
/*       = w*w*y */
/*    x2 = h*derx2 + oldx2; */
/*    x1 = h*x2 + oldx1; */

/* SOLVING LINEAR SYSTEM OF EQUATIONS */

/* LINEAR TEARING VARIABLES AND RESIDUES */
/*         derx2         linresderx2 */

/* TORN LINEAR EQUATIONS */
      derx2 = (w*w*y - ((2*w*D + h*w*w)*oldx2
              + w*w*oldx1)) / (1 + (2*w*D + h*w*w)*h));
/* END OF TORN LINEAR EQUATIONS */

      x2 = h*derx2 + oldx2;
      x1 = h*x2 + oldx1;
/* END OF SYSTEM OF LINEAR SIMULTANEOUS EQUATIONS */

      u = -k*x1;
      residuederx = f(x,u,Time) - derx;
/* END OF TORN NONLINEAR EQUATIONS */

      QRes[0] = residuederx;
      }

/* UPDATE SOLUTION */
      DymNon(QInfRev,QiOpt,QNnl,QSol,QRes,QJac,Qtol,
        Qinfo,QD,QI,PrintEvent,QNLnr,Time,QNLfunc,
        QNLjac,QNLmax,QiErr);

      derx = QSol[0];

      if (QInfRev > 0) goto Iter1;
      if (*QiErr != 0) goto leave;
/* END OF SYSTEM OF NONLINEAR SIMULTANEOUS EQUATIONS */

/* END OF SORTED AND SOLVED EQUATIONS */
```

This code executes considerably faster than the DAE code. The reason is that the number of nonlinear iteration variables has been reduced from three to one. Thereby, the Jacobians are reduced from being matrices of sizes 3×3 to mere scalars, and the number of iterations needed to reach convergence will also be smaller.

Note, that the Newton iteration in the C–code is done using "reverse communication," that is, the residuum is calculated, and afterwards, the nonlinear (Chord–) Newton iteration function `DymNon` is called. When `DymNon` needs a new residuum calculation, the function is left signaling the desired action via variable `QInfRef`, and the code jumps again to the residuum calculation.

MECHATRONICS EXAMPLE

In (Franke and Otter, 1993), a realistically modeled mechatronics system consisting of a six–degree of freedom robot together with its drive trains, motors, actuators, and the electronic control circuitry was described for use as a benchmark problem. In figure 4, a screen dump of this system modeled with Dymola's graphical editor, Dymodraw, is shown.

On the right side of the figure, the six–degree of freedom robot is shown composed of basic mechanical components like joints and bars (Otter *et al.*, 1993). At every joint, a drive train D_i is present. Every such object contains a gearbox (not shown in the figure), a motor, and an actuator, as well as a control system. The elasticity of the gears of the first three joints is modelled by one spring for each gearbox. The elasticity of the last three joints is neglected. Damping and Coulomb friction are considered in every joint. In the upper part of figure 4, the model of the motor and actuator of one joint can be seen. This component is defined, most naturally, as an electrical circuit. Finally, in the lower part of figure 4, the tacho filters and the control system of one drive train are defined in block diagram format. In the left part of figure 4, some component libraries are shown that are used to build up the model.

The model consists of 12 states for the mechanical part of the robot, two states for every gearbox with modeled elasticity, two states for every motor/actuator component, three states for every tacho filter, and three states for every controller. The overall systems has therefore $12 + 3 \cdot 2 + 6 \cdot (2 + 3 + 3) = 66$ states. The model is build up by about 2000 equations.

The system is stiff due to the stiff springs in the gearboxes and due to the "fast" controllers, i.e., due to artificial stiffness. Consequently, an implicit integration algorithm is most appropriate. Available standard stiff system solvers, like LSODAR or DASSLRT, have to perform a Newton iteration on a system of 66 algebraically coupled nonlinear equations, a formidable task. The Jacobian and Hessian of the system are matrices of 4356 elements. An LU–decomposition of the Hessian ($O(n^3)$ Operations, $n = 66$) needs to be performed whenever the Newton iteration does not converge fast enough. A back–substitution to solve the linear equation ($O(n^2)$ Operations) is needed once per Newton iteration step.

By using the (semi–automatic) inline integration procedure explained in the last sections, it is possi-

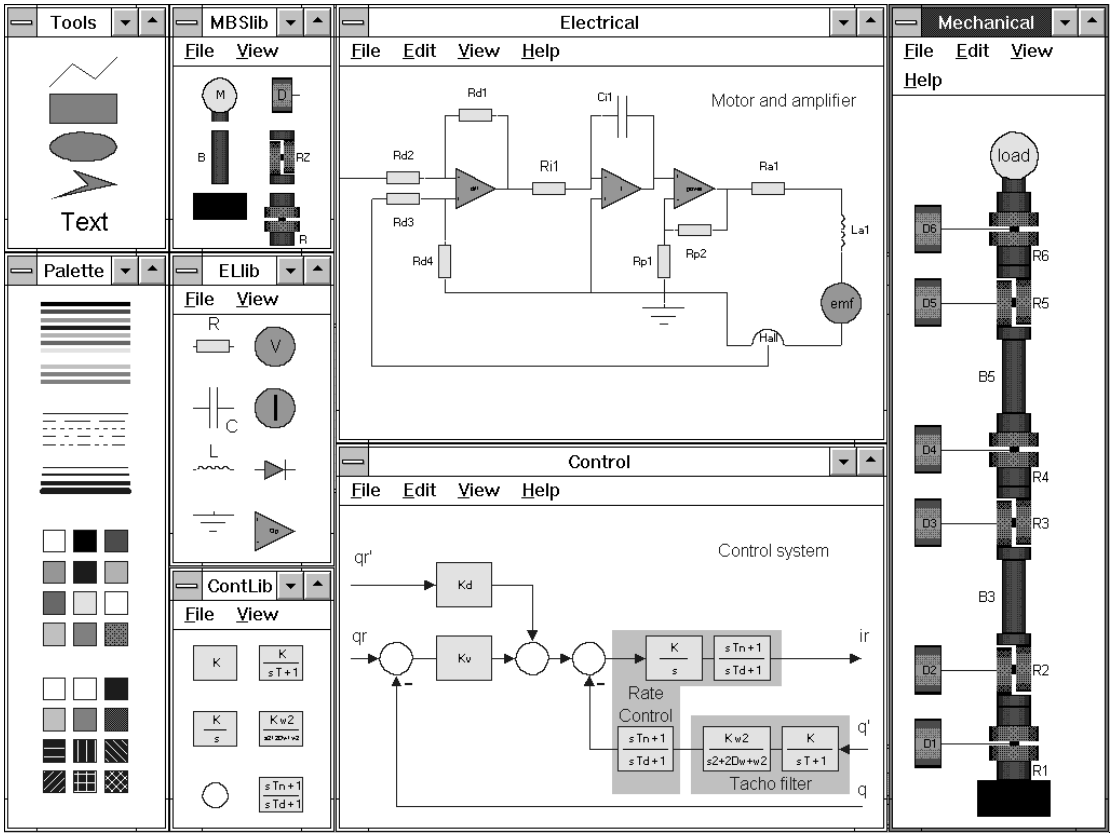

Figure 4: Object–Oriented View of Mechatronic Model

ble to reduce the Newton iteration from 66 equations down to six equations. The main reason for this drastic reduction is that most of the components have linear dynamics (although several of these components are coupled by nonlinear elements, such as limiters).

Let us discuss the reduction process in more detail. The 12 state equations of the robot can be reduced to six nonlinear equations, since the original equations are of second order, and higher–order derivatives are eliminated as explained in (9,10). Therefore, the states of the robot, i.e., the angle and angular velocity of every joint, can be considered known (the angular velocity variables are the tearing variables used for Newton iteration, the angles are computed from the discretization equations).

These known variables enter into the six drive trains that are totally decoupled from each other (provided the angle and angular velocity variables of the robot are known), and which to a great extent have *linear* dynamics. A similar reasoning as the one provided in the section entitled *Tearing* shows that these systems can be fully eliminated, resulting in the inversion of six linear systems of equations containing either $2+2+3+3=10$ (a drive train of the first three joints) or $2+3+3=8$ (a drive train of the last three joints) equations. Since the drive trains are build up by loosely coupled first– or second–order systems, the inversion of a 10×10 system actually breaks down into the inversion of a series of 1×1 and 2×2 systems.

This can easily be seen for the controller part (cf. lower part of figure 4). The angle `q` and the angular velocity `q'` are known and enter from the right into the block diagram of the control system. The first block of the tacho filter is a first order block. Since the input and the discretization are known, the unknown state kann be determined by one division. Afterwards the output can be calculated. The next block is a second–order system. Again, the two states can be determined by solving a system of two equations. In the same way, all other blocks of the control system can be determined, finally leading to the required value of the controller current, `ir`, leaving the control system to the right of the controller.

Note that the above description is just an analysis of what is going on in the translator. In Dymola, the whole procedure is done in a semi–automatic way. Presently, the user has to state in model libraries (from knowledge about the system) which of the tearing variables are associated with linear dynamic systems.

FUTURE RESEARCH

In this paper, we only discussed the inline implementation of BDF algorithms. However, in (Cellier, 1995), it is shown that also implicit Runge–Kutta (IRK) codes can elegantly be inlined. Due to their better accuracy properties (larger asymptotic regions) and better stability properties (even higher–order IRKs can be made L–stable), they represent very attractive alternatives to BDF algorithms for DAE solution. Professional IRK codes implementing e.g. Radau IIa algorithms are available and are widely used (Hairer *et al.*, 1989). Notice that a k–stage fully–implicit IRK algorithm applied to an nth order system requires a Newton iteration on $n \cdot k$ variables, i.e., the systems of equations to be solved are even larger than in the case of the BDF algorithms. For example, the mechatronics problem solved using a 5th order (3–stage) Radau IIa method would call for a Newton iteration on a 198 variable system. For this reason, it is to be expected that inlining will lead to even more spectacular savings when applied to IRK methods.

It should also be much easier, using the inlining technique, to study the behavior of mixed stiff/non–stiff algorithms. It should be possible to separate stiff from non–stiff iteration variables and apply Newton iteration only to the stiff variables, whereas the non–stiff variables are iterated using the much cheaper fixed–point iteration method. However, this has not been attempted yet.

CONCLUSIONS

Inline integration was proposed as a new mixed symbolic/numeric approach for solving differential–algebraic equation systems. Inline integration means to add the state discretization equations symbolically to the model equations at the time of model compilation, thereby symbolically converting the original continuous–time model (differential equation system) to a corresponding discrete–time model (difference equation system). If an implicit integration formula is inlined, the resulting difference equations always contain large blocks of simultaneous, i.e., algebraically coupled, equations. These are then structured for enhanced solution speed using BLT-transformation and tearing. The goal of tearing is to further reduce the sizes of the blocks of simultaneous equations to fairly small sets of equations that can be solved in an efficient manner either symbolically at compile time or numerically at run time.

It was shown that BLT-transformation and appropriate tearing cut right across *all* types of equations. They do not distinguish between *model equations*, i.e., the equations that are extracted from the physical description of the system to be simulated, and *numerical equations*, i.e., equations stemming from the numerical solution technique, such as the state discretization equations. Consequently, it is important to treat both types of equations in the same manner. This makes it necessary to augment the model equations by the state discretization equations symbolically at compile time prior to determining suitable tearing structures.

The effectiveness of inline integration was demonstrated by means of a large mechatronics model consisting of 66 differential and about 2000 algebraic equations. It was possible to reduce the nonlinear system of discretized equations in a semi–automatic fashion from 66 down to six, i.e., by a factor of more than 10.

Inlining has been implemented as a new feature in the modeling language Dymola, an object–oriented modeling tool for large continuous and discontinuous models of physical and engineering systems.

REFERENCES

Andersson, M. (1994), *Object–Oriented Modeling and Simulation of Hybrid Systems*, Ph.D. Dissertation, Report CODEN: LUTFD2/TFRT-1043-SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Andrzejewski, T., H.G. Bock, E. Eich and R. von Schwerin (1993), "Recent Advances in the Numerical Integration of Multibody Systems", *Advanced Multibody System Dynamics*, edited by W. Schiehlen, Kluwer Academic Publishers, pp. 127–151.

Augustin, D.C., 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.

Boston, J.F. (1980), "Inside–out Algorithms for Multicomponent Separation Process Calculations," *Computer Applic. to Chem. Engng*, pp. 135–151.

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

Cardona, A. and M. Gérardin (1993), "Numerical Integration of Second Order Differential–Algebraic Systems in Flexible Mechanism Dynamics", *Proceedings NATO/ASI, Computer–Aided Analysis of Rigid and Flexible Mechanical Systems*, Vol. 1, Troia, Portugal, June 27 – July 9, pp. 165–193.

Cellier, F.E. (1995), *Continuous System Simulation*, Springer–Verlag, New York, to appear.

Cellier, F.E., and H. Elmqvist (1993), "Automated Formula Manipulation Supports Object–Oriented Continuous–System Modeling," *IEEE Control Systems*, **13**(2), pp. 28–38.

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

Elmqvist, H. (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.

Elmqvist, H. (1993), "A Novel Approach to Integrating DAE's," Electronic mail message to F.E. Cellier and M. Otter, April 3.

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

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

Franke, J., and M. Otter (1993), *The Manutec r3 Benchmark Models for the Dynamic Simulation of Robots*, Technical Report TR R101-93, DLR, Institut für Robotik und Systemdynamik, D–82234 Wessling.

Gear, C.W. (1971), *Numerical Initial Value Problems in Ordinary Differential Equations*, Series in Automatic Computation, Prentice-Hall.

Hairer, E., C. Lubich, and M. Roche (1989), *The Numerical Solution of Differential–Algebraic Systems by Runge–Kutta Methods*, Springer–Verlag, Berlin, Germany.

Lubich, C., U. Nowak, U. Pöhle, and C. Engstler (1993), "An Overview of MEXX: Numerical Software for Integration of Multibody Systems", *Advanced Multibody System Dynamics*, edited by W. Schiehlen, Kluwer Academic Publishers, pp. 421–426.

Kron, G. (1962), *Diakoptics — The piecewise Solution of Large-Scale Systems*, MacDonald & Co., London.

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

Otter, M., H. Elmqvist, and F.E. Cellier (1993), "Modeling of Multibody Systems With the Object–Oriented Modeling Language Dymola," *Proceedings NATO/ASI, Computer–Aided Analysis of Rigid and Flexible Mechanical Systems*, Vol. 2, Troia, Portugal, June 27 – July 9, pp. 91–110.

Otter, M. (1995), "Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter", Ph.D.-Dissertation, Fortschritt-Berichte VDI, Reihe 20, Nr. 147, VDI-Verlag Düsseldorf.

Petzold, L.R. (1983), "A description of DASSL: A differential/algebraic system solver," *Scientific Computing*, edited by R.S. Stepleman et al., North–Holland, Amsterdam, pp. 65–68.

Simandl, J., and W.Y. Svrcek (1991), "Extension of the Simultaneous–Solution and Inside–Outside Algorithms to Distillation with Chemical Reactions," *Computer and Chemical Engng*, **15**(5), pp. 337–348.