Generation of Sparse Jacobians for the Function Mock-Up Interface 2.0

J. Åkesson[a,c], W. Braun[d], P. Lindholm[b], B. Bachmann[d]
[a]Lund University, Department of Automatic Control, Lund, Sweden
[b]Lund University, Department of Mathematics, Lund, Sweden
[c]Modelon AB, Lund, Sweden
[d]University of Applied Sciences Bielefeld, Bielefeld, Germany

Abstract

Derivatives, or Jacobians, are commonly required by numerical algorithms. Access to accurate Jacobians often improves the performance and robustness of algorithms, and in addition, efficient implementation of Jacobian computations can reduce the over-all execution time. In this paper, we present methods for computing Jacobians in the context of the Functional Mock-up Interface (FMI), and Modelica. Two prototype implementations, in JModelica.org and OpenModelica are presented and compared in industrial as well as synthetic benchmarks.

Keywords: FMI; Analytic Jacobians; Automatic Differentiation; JModelica.org; OpenModelica;

1 Introduction

Algorithms for solving computational problems numerically often require access to derivatives, or approximations thereof. Examples include simulation algorithms, where implicit integration schemes use derivative information in Newton type algorithms, optimization algorithms, where derivatives are used to compute search directions, and steady-state solvers. The quality of the derivatives typically affects performance and robustness of such algorithms. Often, the execution time is strongly affected by the calculation time of Jacobians.

During the last two years, the Functional Mock-up Interface [1] (FMI) standard has had a strong impact amongst software tools for modeling and simulation. The goal of the standard is to promote model reuse and tool interoperability by providing a tool and language independent exchange format for models in compiled or source code form. Following the introduction of FMI 1.0 in January 2010, the next version of the standard, FMI 2.0, will support sparse Jacobians, in order to enable increased efficiency of algorithms supporting FMI. The target of this extension is to provide derivative information for two different use cases of Functional Mock-up Units (FMUs). The first use case is simulation of a single FMU. In this case, sparse Jacobians for the model equations enable increased efficiency of iterative integration algorithms. The second use case is the composition of multiple FMUs, potentially blended also by elements from a modeling language such as Modelica, where directional derivatives are useful in order to efficiently construct Jacobians for systems of equations spanning several FMUs.

In this paper, we describe methods for generating sparse Jacobians and directional derivatives to fulfill the corresponding requirements of FMI 2.0. The methods are described in the context of compilation of Modelica models into FMUs, although the employed techniques are generally applicable to other model description formats. Two prototype implementations, one in OpenModelica[2] and one in JModelica.org[3] are presented. The implementations of sparse Jacobians in the respective tools are compared based on industrial benchmark models.

The paper is organized as follows. In Section 2, material on FMI, Jacobians and differentiation techniques are provided. Section 3 describes two different implementations of sparse Jacobians in JModelica.org and OpenModelica respectively. Benchmark results are provided in Section 4, and the paper ends with a summary and conclusions in Section 5.

[1]https://fmi-standard.org/

[2]http://www.openmodelica.org

[3]http://www.jmodelica.org

2 Background

2.1 The Functional Mock-up Interface

FMI emerged as a new standard resulting from the ITEA2 project MODELISAR, in 2010. The standard is a response to the industrial need to connect different environments for modeling, simulation and control system design. Commonly, different tools are used for different applications, whereas simulation analysis at the system integration level requires tools to be connected. FMI provides the means to perform such integrated simulation analysis.

FMI specifies an XML format for model interface information and a C API for model execution. The XML format, specified by an XML schema, contains information about model variables, including names, units and types, as well as model meta data. The C API, on the other hand, contains C functions for data management, e.g., setting and retrieving parameter values, and evaluation of the model equations. The implementation of the C API may be provided in source code format, or more commonly as a compiled dynamically linked library.

FMI comes in two different flavors: FMI for Model Exchange (FMI-ME) [2] and FMI for Co-Simulation (FMI-CS) [3]. FMI-ME exposes a hybrid Ordinary Differential Equation (ODE), which may integrated stand-alone or which may be incorporated in a composite dynamic model in a simulation environment. The FMI-ME C API exposes functions for computation of the derivatives of the ODE, and accordingly, in FMI-ME the integration algorithm is provided by the importing application. FMI-CS, on the other hand, specifies that the integration algorithm is included in the FMU, and the FMU-CS C API provides functions for integrating the dynamics of the contained ODE for a specified period of time.

The FMI standard is supported by several modeling and simulation tools, including Dymola, SimulationX, JModelica.org and OpenModelica. Also, there are FMI interfaces to MATLAB, National Instruments Veristand and several additional tools.

FMI 2.0 is a unification of the Model Exchange and Co-simulation standards and contains several improvements. One of those are the sparse Jacobians, which are also topic of this paper. The sparse Jacobian interface in FMI 2.0 consists of three different parts:

- A C API function for evaluation of directional derivatives of the model equations.
- A C API function for evaluation of sparse Jacobian matrices corresponding to the ODE representation of an FMU.
- A section in the XML document contained in an FMU providing the incidence pattern for the Jacobian matrices.

In this paper, algorithms for generating this functionality are discussed.

2.2 Causalization of DAEs

In the first step of the compilation process in a Modelica tool chain, a compiler front-end transforms Modelica source code into a flat representation, consisting essentially of lists of variables, functions, equations and algorithms. Based on this model representation, symbolic operations such as alias elimination and index reduction are applied, in order to reduce the size of the model and to ensure that the resulting Differential Algebraic Equation (DAE) is of index 1. In this section, we outline the following steps that are of particular relevance for the generation of Jacobians. In particular, the causalization procedure, i.e., transformation of an index-1 DAE into an equivalent ODE, as required by the FMI standard, is discussed.

FMI specifies Jacobians and directional derivatives with respect to the continuous model equations. Therefore, without lack of generality, and for clarity of the presentation, only the continuous part of the DAE is considered in the following.

We consider index-1 DAEs in form of

$$\begin{aligned} F(\dot{x}(t), x(t), u(t), w(t)) = 0, \quad t \in [t_0, t_f] \\ x(0) = x_0 \end{aligned} \tag{1}$$

where $\dot{x}(t) \in R^{n_x}$ are the state derivatives, $x(t) \in R^{n_x}$ is the state, $u(t) \in R^{n_u}$ are the inputs and $w(t) \in R^{n_w}$ are the vector of algebraic variables. The initial conditions of DAE state is given by x_0. Introducing $z = (\dot{x}\ w)$, denoting the unknowns of the DAE, and $v = (x\ u)$, denoting the known variables, the DAE written

$$F(z, v) = 0 \tag{2}$$

The conceptual idea of DAE causalization commonly used in Modelica tools is then to compute the inverse relationship of F

$$z = G(v), \tag{3}$$

and the ODE may then be written

$$\begin{aligned} \dot{x} = f(x, u) \\ y = h(x, u) \end{aligned} \tag{4}$$

where y are the outputs of the system. Note that the algebraic variables are considered to be internal to the ODE in this representation. In general, there is no closed expression for the functions f and g, but rather, iterative techniques, e.g., Newton's method, is employed to solve algebraic loops for z.

Modelica models are typically of large scale but sparse in the sense that each model equation contains references only to a small number of equations. In order to exploit this structure, graph algorithms can be employed. Two commonly used algorithms that are used for this purpose are *matching algorithms*, e.g., the Hopcroft Karp algorithm, and Tarjan's algorithms for computing *strong components*, [4]. The result of Tarjan's algorithm is then used to permute the variables and equations of the DAE into Block Lower Triangular (BLT) form.

Let us consider a DAE with five equations and five unknowns, i.e., $F \in R^5$ and $z \in R^5$, where the DAE equations are given by

$$\begin{aligned} F_1(z_1, z_5, v) &= 0 \\ F_2(z_3, v) &= 0 \\ F_3(z_1, z_2, z_3, z_4, v) &= 0 \\ F_4(z_1, z_3, z_5, v) &= 0 \\ F_5(z_2, z_5, v) &= 0 \end{aligned} \tag{5}$$

Note that the variables $v = [x, u]$ are known and need not be considered in the following analysis. The dependence of the z-variables can be shown in the following incidence matrix,

$$\begin{array}{c|ccccc} & z_1 & z_2 & z_3 & z_4 & z_5 \\ \hline F_1 & * & 0 & 0 & 0 & * \\ F_2 & 0 & 0 & * & 0 & 0 \\ F_3 & * & * & * & * & 0 \\ F_4 & * & 0 & * & 0 & * \\ F_5 & 0 & * & 0 & 0 & * \end{array} \tag{6}$$

A * in the incidence matrix at row i and column j denotes that the residual function F_i contains a reference to the variable z_j. Application of the BLT procedure, now yields the following DAE system

$$\begin{array}{c|ccccc} & z_3 & z_1 & z_5 & z_2 & z_4 \\ \hline F_2 & 1 & 0 & 0 & 0 & 0 \\ F_4 & 1 & 1 & 1 & 0 & 0 \\ F_1 & 0 & 1 & 1 & 0 & 0 \\ F_5 & 0 & 0 & 1 & 1 & 0 \\ F_3 & 1 & 1 & 0 & 1 & 1 \end{array} \tag{7}$$

The implicit DAE system (5) is now given by a sequence of assignment statements and implicit systems of equations

$$\begin{aligned} \bar{z}_1 &:= g_1(v) \\ \bar{F}_2(\bar{z}_1, \bar{z}_2, v) &= 0 \\ \bar{F}_3(\bar{z}_2, \bar{z}_3, v) &= 0 \\ \bar{z}_4 &:= g_4(\bar{z}_1, \bar{z}_2, \bar{z}_3, v) \end{aligned} \tag{8}$$

where $\bar{z}_1 = z_3$, $\bar{z}_2 = (z_1\ z_5)^T$, $\bar{z}_3 = z_2$, $\bar{z}_4 = z_4$. The functions g_1 and g_2 corresponds to explicit solutions of the corresponding DAE equations, whereas $\bar{F}_2 = (F_4\ F_1)^T$ and $\bar{F}_3 = F_5$ corresponds to implicit (systems of) equations that require iteration. It is typical for Modelica models to contain only a small number of implicit systems of equations and a large number of trivial, e.g., linear equations that may be solved symbolically.

For a general DAE, the BLT procedure results in a sequence of scalar and non-scalar equation blocks on the form

$$\begin{aligned} \bar{F}_1(\bar{z}_1, v) &= 0 \\ &\vdots \\ \bar{F}_i(\bar{z}_1, ..., z_i, v) &= 0 \\ &\vdots \\ \bar{F}_b(\bar{z}_1, ..., z_b, v) &= 0 \end{aligned} \tag{9}$$

where the unknown of each residual $\bar{F}_i$ is $\bar{z}_i$. Further, some of the residual functions may be solved explicitly by symbolic manipulation and the remaining blocks needs the to be solved by iterative methods.

Computation of the sequence of solved and non-solved blocks (9), given values of the known variables in v then produces the corresponding state derivative and algebraic vectors contained in z. Accordingly, the DAE has been causalized in to an ODE on the form (4).

2.3 Computation of Jacobians

The Jacobian of a vector valued function $f(x) \in R^m$, $x \in R^n$ is given by

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} \tag{10}$$

A useful tool when computing Jacobians is directional derivatives. The directional derivative of a vector valued function $f(x)$ is defined by

$$df = \frac{\partial f}{\partial x} \cdot dx, \tag{11}$$

where $dx \in R^n$ represents the direction in which the directional derivative, denoted $df \in R^m$, is evaluated. dx is also referred to as a seed vector.

In the following, directional derivatives will be used extensively to construct Jacobians. A straight forward, although naive, approach to construct a Jacobian from directional derivative evaluations is as follows. Using the identity matrix I of dimension n, and the unit vectors $e_1 \dots e_n$ we have that

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial x} I = \frac{\partial f}{\partial x} \begin{pmatrix} e_1, & \dots & e_n \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial x} \cdot e_1 & \dots & \frac{\partial f}{\partial x} \cdot e_n \end{pmatrix}. \quad (12)$$

Using this relation, a Jacobian with n columns may be constructed from n evaluations of directional derivatives. In Section 2.7, an overview of methods to explore sparsity to improve efficiency in this respect will be given.

There are three widely used methods for computing Jacobians, namely finite difference methods, symbolic differentiation and automatic (or algorithmic) differentiation.

2.4 Finite Difference Approximation

In the finite difference method, a numerical approximation of the directional derivative of a vector valued function f is calculated using the formula

$$\frac{\partial f(x)}{\partial x} \cdot e_i = \frac{f(x+e_i h) - f(x)}{h}. \quad (13)$$

where h is the increment. On one hand, even if the increment is chosen optimal in nature of that method is an accuracy error ε, which is the sum of $\varepsilon_t + \varepsilon_r$ where ε_t is the truncation error and ε_r the round-off error. The truncation error ε_t $|h\ddot{f}(x)|$ is the result of the Taylor-series truncation. The round-off error ε_r $\varepsilon_f |f(x)/h|$ where ε_f is the fractional accuracy $\varepsilon_f \geq \varepsilon_m$ depends on machine accuracy ε_m. On the other hand, it is easy to implement and also almost applicable.

2.5 Symbolic Differentiation

In general the "calculus" of symbolic derivatives is done by difference quotients. where the derivative of a function is the limit

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \quad (14)$$

difference quotients. This is also the way the basic differentiation rules are found. From a practical view the "calculus" of the symbolic derivatives is done by applying basic differentiation rules and table of derivatives for common functions on the expressions to find the formulas for the derivatives. Since a Modelica model results during the compilation in symbolic expressions which are manipulated to simplify the original system. So it is quite typical for a Modelica Tool to use symbolical methods also for the differentiation. Finding the symbolic formula may take time, space and a symbolic kernel for simplifications, but once determined it's fast to evaluate them [7]. A further drawback is that symbolic differentiation is not applicable on algorithms (with for-loops and branches).

2.6 Automatic Differentiation

Automatic Differentiation (AD) is a method for computing derivatives with machine precision, which is applicable to expressions as well as algorithmic functions [1]. The key idea in AD techniques is to propagate derivative information through a sequence of atomic operations, which is represented by an *expression graph*. Computation of a sequence of AD operations results in the evaluation of a directional derivative with respect to a given seed vector.

There are two different modes of operation of AD—forward and reverse. The forward mode AD is conceptually simple, and is based on forward propagation of values and derivatives through an expression graph. The result of a forward AD sweep is a vector corresponding to the Jacobian multiplied by the seed vector. Commonly, Jacobian matrices are constructed from a number of forward AD evaluations.

The reverse AD technique is more involved than the forward mode, and includes a forward and a backward evaluation sweep over the expression graph, and the result is a vector corresponding to the seed vector multiplied by the Jacobians. This mode of operation is particularly useful in the case of scalar functions that depends on many independent variables—in this case, reverse AD is referred to the cheap gradient computation. Reverse AD is also commonly used to construct higher-order derivatives, e.g., Hessian matrices in optimization applications.

Implementation of AD tools comes two different flavors: Operator Overloading (OO) and Source Code Transformation (SCT). In OO tools, the expression graph is represented by data structures that are repeatedly traversed during forward and reverse mode evaluations. This strategy has been popularized by tools such as CppAD[4] and ADOL-C[5] which both enable AD to be applied to C code with minor modifications. Tools in this category are typically based on operator overloading, e.g., in C++, to construct a data structure referred to as a *tape*, which is then used as a basis for derivative computations. Tools based on the SCT approach, on the other hand, generate code that, when executed, compute derivatives. The ADIFOR[6] package falls into this category.

In this paper, forward mode AD using the SCT technique will be used. The remainder of this section will therefore focus on explaining this methods.

A key to understanding forward AD, is the observation that expressions can be evaluated, and differentiated, by considering a sequence of atomic operations. The elementary arithmetic operations can be differentiated by applying the derivation rules

$$\begin{aligned}\frac{d}{dx}(u(x)\pm v(x)) &= \frac{du}{dx}\pm\frac{dv}{dx}\\ \frac{d}{dx}(u(x)v(x)) &= u(x)\frac{dv}{dx}+v(x)\frac{du}{dx}\\ \frac{d}{dx}\left(\frac{u}{v}\right) &= \frac{v(x)\frac{du}{dx}-u(x)\frac{dv}{dx}}{v(x)^2}\end{aligned}$$

In addition, the chain rule

$$\frac{d}{dx}\phi(u(x))=\frac{d\phi}{du}\frac{du}{dx}$$

applies to the elementary arithmetic functions, such as sin, cos etc.

In the following example, we illustrate how these building blocks are used to apply the forward AD technique. We consider the function

$$f(x_1,x_2)=x_1\cdot x_2+\sin(x_1), \qquad (15)$$

for which we would like to compute the directional derivative according to relation (11). Assuming the seed vector $dx=(1\ 0)^T$, it follows that

$$df=\frac{\partial f}{\partial x}dx=\begin{pmatrix}\frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2}\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix}=\frac{\partial f}{\partial x_1}. \qquad (16)$$

Using the seed vector in (16), $f(x)$ will be differentiated with respect to x_1.

The expression graph corresponding to the function in (15) is shown in Figure 1.

In the figure, the leaves represent the independent variables and the root node represents the function itself.

[4]http://www.coin-or.org/CppAD/

[5]http://www.coin-or.org/projects/ADOL-C.xml

[6]http://www.mcs.anl.gov/research/projects/adifor/

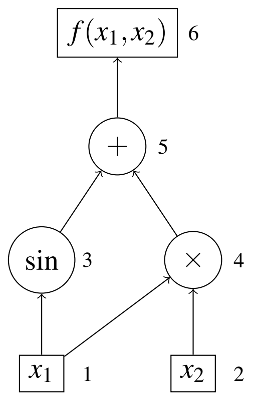

Figure 1: Expression graph of the function (15)

A forward AD sweep is performed as follows. The computation sequence starts at the independent variables. Intermediate variables, v_i:s, are introduced to hold the value of each node, and in addition, variables for the derivative values of each node, d_i, are introduced. The expression of a particular variable v_i is given by the corresponding node type, i.e., arithmetic operation, and the derivative value, d_i, is given by differentiation of the same operation. Application of this procedure to the function (15) gives the following sequence of operations.

$$\begin{aligned}v_1 &:= x_1\\ d_1 &:= dx_1\\ v_2 &:= x_2\\ d_2 &:= dx_2\\ v_3 &:= sin(v_1)\\ d_3 &:= d_1\cdot cos(v_1)\\ v_4 &:= v_1\cdot v_2\\ d_4 &:= d_1\cdot v_2+v_1\cdot d_2\\ v_5 &:= v_3+v_4\\ d_5 &:= d_3+d_4\\ v_6 &:= v_5\\ d_6 &:= d_5\end{aligned}$$

The variable v_6 now holds the value of the function itself and d_6 holds the value of the directional derivative. Note that the evaluation is done for particular values of the independent variables, in this case x_1 and x_2, and seed values, dx_1 and dx_2. Note that auxiliary variables v_1, v_2, d_1 and d_2 are introduced here for clarity.

2.7 Exploiting Sparsity

Modelica models, also after the causalization procedure described above, are often sparse, i.e., each equation of a model depends only on a fraction of the total number of variables. Exploiting sparsity of Modelica models can be done in two different contexts. Firstly, the efficiency of computation of Jacobian matrices based on directional derivative evaluations can be much improved by considering sparsity. This strategy is called *compression* and will be described briefly in this section. Secondly, a simulation environment importing an FMU providing sparse Jacobians may utilize this information to improve the performance of numerical algorithms. A typical example of such algorithms are sparse linear solvers, e.g., UMFPACK[7], CSparse[8] and PARDISO[9]. This usage is, however, not related to the procedures required to generate Jacobians, and it is therefore beyond the scope of this paper.

As noted above, a naive method for evaluation directional derivatives to generate Jacobian matrices is to simply make one such evaluation for each column of the Jacobian, with seed vectors corresponding to the unit vectors of appropriate dimension. If the Jacobian is sparse, however, the number of evaluations can be drastically reduced, by observing that several columns can be computed in a single directional derivative evaluation if the sparsity patterns of these columns do not overlap. As an example, consider the incidence matrix (6). Here, we note that columns four and five does not contain overlapping entries, and they can therefore be computed by one single directional derivative evaluation with the seed vector chosen as the sum of the corresponding unit vectors. Note also that this strategy is applicable to all three differentiation methods described above: finite differences, AD and symbolic differentiation.

While this strategy is simple to implement, computing a column grouping of minimal size is well known to be an NP-hard problem—this problem corresponds precisely to the graph coloring problem [5, 6]. There are, however, efficient algorithms capable of computing practically useful approximations of the optimal solutions. Specific algorithms will be discussed in Section 3.

[7]http://www.cise.ufl.edu/research/sparse/umfpack/

[8]http://people.sc.fsu.edu/~jburkardt/c_src/csparse/csparse.html

[9]http://www.pardiso-project.org/

3 Computation of Jacobians for Modelica Models

In Section 2.2, it was shown how a DAE is transformed into an ODE by means of the BLT transformation. In this section, methods for computing the Jacobians of the resulting ODE (4) are presented. We consider

$$\frac{\partial z}{\partial v} = \begin{pmatrix} \frac{\partial \dot{x}}{\partial v} \\ \frac{\partial y}{\partial v} \end{pmatrix} = \begin{pmatrix} \frac{\partial \dot{x}}{\partial x} & \frac{\partial \dot{x}}{\partial u} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial u} \end{pmatrix} = \left(\begin{array}{c|c} A & B \\ C & D \end{array}\right) \tag{17}$$

In this section, we present two methods for computing the matrices A, B, C and D by means of directional derivatives. One of the methods, which is implemented in JModelica.org, relies on a forward AD implementation in an SCT setting, whereas the other method, which is implemented in OpenModelica, relies on symbolic differentiation and symbolic expression simplification. In addition, an algorithm for computing the sparsity pattern of the Jacobian matrices, which is common for both methods, is presented.

The key idea in this section is the following. Differentiating the DAE (2) yields the relation

$$\frac{\partial F}{\partial z} dz + \frac{\partial F}{\partial v} dv = 0, \tag{18}$$

where dv is the input seed vector and dz works as the directional derivative of the relation (3) with respect to the direction dv. By solving the system of equations (18) for a particular seed dv, the directional derivative of the DAE is obtained. It is important to note that the system of equations to be solved is linear in the unknowns, dz, and thus does not require iteration.

The Jacobian matrices are then constructed from repeated evaluation of directional derivatives. In addition, coloring algorithms and compression is used to reduce the number of directional derivative evaluations in both implementations.

Evaluation of Jacobians based on the compression of the columns requires access to sparsity pattern, as stated in Section 2.7. The determination of the sparsity pattern for a Modelica model could be done by means of graph theory. Since the non-zero values in a Jacobian expresses which output variable has a connection to which input variable. Thus the determination problem could be formulated as a st-connectivity problem in a directed graph, where input variables are the sources and the output variables are the sinks. The st-connectivity is a decision problem that asks if the vertex t is reachable from the vertex s, particular which output variable is connected to which input variable. Specific algorithms for this purpose will be discussed below.

3.1 Implementation of Directional Derivatives in JModelica.org

The performance of the approach outlined above can be improved significantly by exploiting the BLT structure described in Section 2.2. In particular, forward AD may be applied directly to the sequence of computations given in (9). In the implementation in JModelica.org, C code corresponding to a forward AD sweep over the sequence of BLT blocks is generated. The symbolic expression graphs in the compiler is a basis for the code generation. As noted in Section 2.2, there are two kinds of blocks produced by the BLT transformation, i.e solved equation blocks and non-solved equation blocks requiring iterative numerical solution. Below, we explain how directional derivatives are propagated in these two cases.

3.1.1 Propagation of Directional Derivatives in Equation Blocks

For blocks corresponding to solved equation blocks of the form

$$\bar{z}_i := g_i(\bar{z}_1, \dots, \bar{z}_{i-1}, v) \tag{19}$$

it is straight forward to apply the forward AD approach. In this case, AD code is simply generated basing on the expression graph for g_i, in order to produce the directional derivative

$$d\bar{z}_i = \frac{\partial g_i}{\partial \bar{z}_1} d\bar{z}_1 + \dots + \frac{\partial g_i}{\partial \bar{z}_{1-i}} d\bar{z}_{i-1} + \frac{\partial g_i}{\partial v} dv. \tag{20}$$

Note that the input seed dv and the directional derivatives for previous blocks, $\bar{z}_i, \dots \bar{z}_{i-1}$ are known at this point in the computation sequence. Commonly, the expression g_i does not depend on all previous vectors of unknowns, $\bar{z}_1, \dots, \bar{z}_{i-1}$, a property which is exploited in the implementation.

For a block corresponding to a system of equations, the block residual is given by

$$\bar{F}_i(\bar{z}_1, \dots, z_i, v) = 0. \tag{21}$$

In order to compute the directional derivative, $d\bar{z}_i$, for such a block, the residual equation is differentiated to yield

$$\frac{\partial \bar{F}_i}{\partial \bar{z}_1} d\bar{z}_1 + \dots + \frac{\partial \bar{F}_i}{\partial \bar{z}_i} d\bar{z}_i + \frac{\partial \bar{F}_i}{\partial v} dv = 0 \tag{22}$$

which in turn gives the linear system

$$\frac{\partial \bar{F}_i}{\partial \bar{z}_i} d\bar{z}_i = -\sum_{k=1}^{i-1} \frac{\partial \bar{F}_i}{\partial \bar{z}_k} d\bar{z}_k - \frac{\partial \bar{F}_i}{\partial v} dv \tag{23}$$

to be solved for $d\bar{z}_i$. All Jacobians in this relation are generated to C code using forward AD. Note that the system Jacobian of the linear system (23) is provided also to the Newton solver that computes the solution of the system of equations (21). Therefore, this code is reused in the computation of the directional derivative of the block.

3.1.2 Computation of Sparsity Patterns

Computation of sparsity patterns for the Jacobian matrices A, B, C and D is a non-trivial problem, because of the sequence of operations required to compute the state derivatives x and the algebraic variables w. In comparison, computation of the Jacobian matrix of a DAE system (2) is straightforward and can be done by simply collecting references to unknown variables in each residual equation. As noted above, the problem of computing sparsity patterns for the ODE Jacobian is a connectivity problem, where the dependencies of the dependent variables z of the independent variables contained in v need to be computed.

The BLT form of the DAE offers means to compute the required sparsity patterns for the ODE Jacobians. While the general form of a block in the BLT sequence is

$$\bar{F}_i(\bar{z}_1, ..., z_i, v) = 0, \tag{24}$$

particular blocks typically do not depend on all variables in $z_1, \dots z_i$ and in v. In order to reflect this situation, we introduce the notation

$$\tilde{F}_i(\tilde{z}_i, z_i, \tilde{v}_i) = 0, \tag{25}$$

where $\tilde{z}_i$ contains the variables in the z vector upon which the equation block residual $\tilde{F}_i$ depends. $\tilde{v}_i$ is defined correspondingly. As a first approximation, which will be relaxed in the following, we assume that all variables solved for in the block i, i.e., z_i, depends on all variables in $\tilde{z}_i$ and in $\tilde{v}_i$. Clearly, this relationship defines the direct dependency of z_i on $\tilde{v}_i$. Now, the dependency of z_i on the variables contained in $\tilde{v}_1, \dots, \tilde{v}_{i-1}$ is given implicitly by $\tilde{z}_i$. The connectivity graph of the BLT form reveals these dependencies. Edges in this graph corresponds to non-zero entries in the lower left part of the transformed incidence matrix, below the block diagonal. In the connectivity graph, dependency information is propagated top-down in the sequence of blocks. For each block, the complete set of variables in v upon which the block depends is collected from the predecessors in the dependency graph.

For a block consisting of a system of equations, the assumption that all variables solved for in the block,

z_i, depends on all variables in $\tilde{z}_i$ may lead to an over-estimation of the sparsity pattern. Specifically, since the sparsity pattern of the inverse of a sparse matrix may also be sparse, the computation may result in non-zero entries which are in fact structural zeros. In order to take this into account, the sparsity pattern of the inverse of the corresponding block Jacobian may be computed, [8]. The result of this analysis is then taken into account when variable dependencies are computed. Note that this analysis remains to be implemented in JModelica.org

3.2 Implementation of Directional Derivatives in OpenModelica

The directional derivatives in OpenModelica are generated basically by setup a new symbolic equation system inside the OMC with the differentiated equations. This system contains the desired partial derivatives $d\bar{z}$ as unknowns, the seed vector $d\bar{v}$ and all other variables from the original system are considered as known. The resulting equation system is the desired one as in equation (18).

This approach differs from the previously published procedure (see [10]), in a way that now each equation is derived only once. This leads to linearity in the compilation time and in the generated code size.

All methods mentioned in section 2 are used for the differentiation of the original system. Equations are differentiated symbolically, algorithm sections and Modelica functions without an derivative annotation are differentiate by the forward AD approach and external functions, where nothing else is possible, are differentiated numerically.

The generated equation system is then optimized like the original system. In detail it is transformed to an explicit form with the BLT machinery of OpenModelica, further expression-based simplification are done and some common sub-expressions are filtered. The resulting equation system is then written to the C-Code.

For the purpose of generating the four matrices in (17) for each matrix one new equation system is generated with the corresponding variables. Note therefore the original system is filtered for the necessary equations.

The exploration of the sparsity pattern for a fast evaluation of the compressed Jacobians is applied on the generated directional derivatives. A detailed description of the algorithms used for that task in OpenModelica can be found in [9].

3.3 Comparison of Implementations

The implementations in OpenModelica and in JModelica.org share common characteristics, but there are also differences. Both algorithms are based on generation of C code that evaluates directional derivatives, which in turn are used to compute Jacobians. Also, both algorithms rely on compression for reducing the number of directional derivative evaluations. The computation of sparsity patterns for the ODE Jacobians also proceeds in the same manner.

The main difference between the implementations is rather the way in which the directional derivatives are generated. In the JModelica.org implementation, the same BLT structure as for the underlying ODE is used. Code generation is done by traversing the BLT structure in a separate code generation pass and forward AD code is then generated for solved equations and systems of equations, as described in 3.1. In the OpenModelica implementation, on the other hand, a new data structure containing all model equations in symbolically differentiated form is first constructed. The symbolic kernel of the compiler is then invoked to simplify the differentiated equations, and a new BLT structure is computed prior to code generation.

Both approaches have advantages and disadvantages. In the JModelica.org implementation, no new data structures are created, which reduces memory consumption. Also, since the same BLT structure as for the underlying ODE is used, Jacobians for systems of equations corresponding to algebraic loops are generated. These, in turn are useful also in case of applying iterative techniques to solve algebraic loops. The main advantage of the OpenModelica implementation is that symbolic simplifications done by the compiler can yield simpler code that is faster to execute. Also, since a new BLT computation is done, properties of the new, differentiated system of equations may be explored in order to further speed up Jacobian computations.

4 Benchmarks

Three different aspects are considered in the benchmarks presented in this section, namely, *i)* model compilation time, *ii)* generated code size, and *iii)* Jacobian evaluation time. In the case of model compilation time, both the time spent in the respective Modelica compilers, OpenModelica and JModelica.org, and the time spent in the C compiler, `gcc` in both cases, when compiling the generated code is measured. This measure seems to be the most interesting for the user, since both phases are included in the model compilation time from a user's perspective. As for the size of the generated code, only the size of the code that is generated by the Modelica compilers is measured, i.e., no code originating from run-time systems or similar is included. Finally, the time for 1000 Jacobian evaluations is measured and the mean evaluation times are reported. In all benchmarks, the system Jacobian, i.e., the Jacobian of the derivatives with respect to the states, is evaluated.

It is worth noting that the benchmarks in this section does not only reflect the particular details of the respective Jacobian evaluation strategies. In particular, the measurements are biased by other code optimization strategies in the compilers, including alias elimination, symbolic processing, tearing, and the efficiency of non-linear solvers used to solve algebraic loops. In addition, the compilation time measurements, the optimization and debugging flags supplied to the respective C compilers influence the result.

All measurements in this paper are performed on a 64-bits architecture computer having one Intel Q9550@2.83GHz CPU and 16 GB of RAM. It runs Ubuntu 12.04 Linux, kernel 3.2.0-25.

4.1 Combined Cycle Power Plant

The first benchmark is a model of a combined cycle power plant model, see Figure 2. The model contains equation-based implementations of the thermodynamic functions for water and steam, which in turn are used in the components corresponding to pipes and and the boiler. The model also contains components for the economizer, the super heater, as well as the gas and steam turbines. The model has 10 states and 131 equations. For additional details on the model, see [11].

The benchmark results are shown in Table 1. As can be seen, the model compilation times and the file sizes are similar. Both implementations obtained six colors for the Jacobian, i.e., 6 directional derivative evaluations were required to compute the Jacobian. The Jacobian evaluation time does, however, differ in a way that the OpenModelica implementation performs faster.

4.2 Synthetic Benchmarks

In order to analyze the scalability properties of the respective implementations, synthetic benchmark models were automatically generated. The underlying assumption of the synthetic models is that a single Modelica equation contains references to fixed maximum number of variables, a number which does not increase with model size. This assumption is realistic, given that Modelica models are typically constructed from a large number of simple component models, where the equations in each component are local in the sense that they refer mainly variables in the same, or neighboring, components. Another important feature of Modelica models are algebraic loops, or implicit systems of equations, which require iterative solution techniques. Therefore, the synthetic benchmark models contain implicit systems of equations, composed from linear and non-linear terms, in the form of *sin* functions, terms.

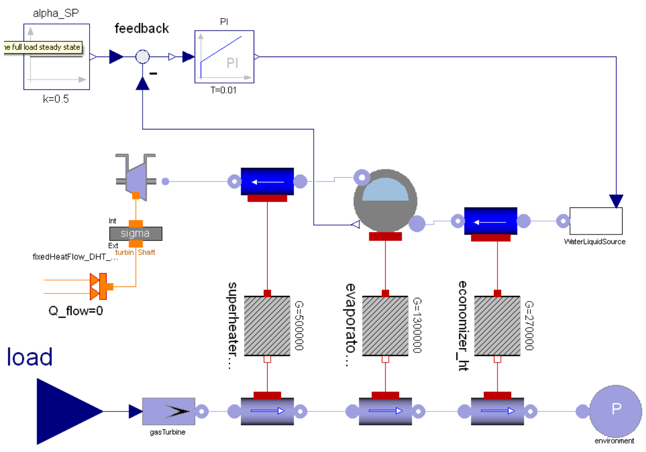

Figure 2: Modelica component diagram for a combined cycle power plant.

Table 1: Benchmark results for combined cycle power plant.

	Generation [s]		Code size [kB]		Jac eval
Tool	No Jac	Jac	No Jac	Jac	time[ms]
OM	2.98	3.87	519	711	0.018
JM	3.64	5.92	266	456	0.090

Three suits of benchmark models were constructed, using different assumptions on the number of variable references in a single equation. This aspect was quantified by the sizes of the implicit systems of equations, where sizes of two, four and eight, respectively, were used to generate the benchmark models. Within each suite of benchmark models, four different models of increasing size were constructed, essentially by doubling the number of variables while keeping the size of all the implicit equation systems constant. For detailed statistics and structural analysis of the models

Table 2: Statistics and structural analysis of the synthetic models. #N denotes the number of variables, #N-z. denotes the number of reported non-zero elements and #Col. denotes the number of colors resulting from the coloring algorithms. #N-z. and #Col. are equal in both implementations.

	#N	#States	#Alg. loops	#N-z.	#Col.
1-1	22	4	9	7	2
1-2	42	8	17	21	4
1-3	82	16	33	47	4
1-4	162	32	65	104	4
2-1	40	4	9	7	2
2-2	76	8	17	21	4
2-3	148	16	33	53	4
2-4	292	32	65	117	4
3-1	76	4	9	7	2
3-2	144	8	17	21	4
3-3	280	16	33	53	4
3-4	552	32	65	117	4

consider table 2. Note that for both implementations, the number of non-zero elements and the number of colors produced by the respective coloring algorithms are equal.

Table 3: Benchmarks of synthetic models for OpenModelica

	Generation [s]		Code size [kB]		Jac. eval.
	No Jac.	Jac.	No Jac.	Jac.	time [ms]
1-1	0.57	1.3	41	121	0.008
1-2	0.87	2.0	72	225	0.033
1-3	1.51	3.7	134	435	0.068
1-4	2.82	7.4	260	860	0.142
2-1	0.88	2.1	64	208	0.017
2-2	1.51	4.1	114	393	0.067
2-3	2.75	7.5	218	781	0.144
2-4	5.36	15.5	429	1569	0.308
3-1	2.22	6.6	117	457	0.048
3-2	4.20	13.7	219	889	0.198
3-3	8.45	27.7	432	1789	0.421
3-4	17.02	56.9	857	3583	0.873

The results in terms of model compilation time, generated code size, and Jacobian evaluation time for the different models are shown in Tables 3 and 4, for the OpenModelica and the JModelica.org implementations respectively. Figures 3, 4, and 5 depict the corresponding results graphically. Each curve corresponds to one benchmark suite. As can be seen from the tables, all three measures exhibit essentially linear complexity for a fixed size of the algebraic loops. This result is the key to scalability of the methods. The smallest model in each benchmark suite deviates from the linear trend for Jacobian evaluation time, which is due to the fact that fewer colors are needed in these cases.

Table 4: Benchmarks of synthetic models for JModelica.org

	Generation [s]		Code size [kB]		Jac. eval.
	No Jac.	Jac.	No Jac.	Jac.	time [ms]
1-1	1.02	1.88	36	138	0.037
1-2	1.24	3.16	54	247	0.089
1-3	1.78	5.71	93	484	0.163
1-4	3.16	10.71	171	957	0.316
2-1	1.37	4.39	61	388	0.104
2-2	2.04	8.17	102	737	0.334
2-3	3.44	15.35	187	1435	0.673
2-4	6.42	31.52	360	2843	1.269
3-1	3.05	15.65	146	1371	0.558
3-2	5.28	30.33	264	2581	2.078
3-3	9.93	63.49	511	5146	4.027
3-4	19.66	136.05	1009	10315	8.827

While model compilation time and generated code size without Jacobians are similar in all cases for OpenModelica and JModelica.org, the corresponding numbers with Jacobians differ. The difference in code size is due to the fact that JModelica.org relies on generation of forward AD code, without simplifications, which results in verbose code. Also, inherent in the forward AD strategy is that both the model equations in their original form and the directional derivatives are evaluated simultaneously. In comparison, the OpenModelica implementation differentiates the equations symbolically and then applies symbolic simplification. In this case, the resulting expressions that are generated are simpler, and also, no additional code is generated for the original model equations.

In terms of execution speed, the OpenModelica implementation performs faster. The main reason for this is that the application of forward AD in the JModelica.org implementation results in more verbose code, and also the model equations, along with the directional derivatives, are evaluated.

It is worth noting that either the effect of different versions of LAPACK/BLAS, used to solve linear systems in both implementations, nor the the influence of different compiler optimization and debugging flags have been considered in the benchmarks. Rather, the performance experienced by users has been reported.

Both implementations may be further optimized in these respects in order to improve compilation and execution times. Therefore, the reported benchmarks do not solely measure the efficiency of the respective methods described in the paper, but are rather biased with the details of the particular implementations.

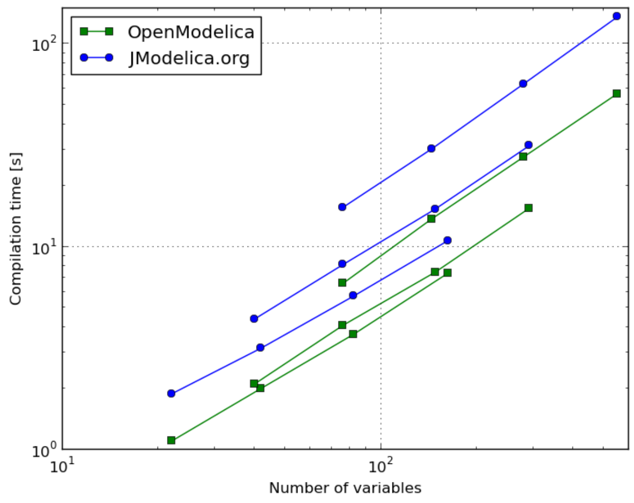

Figure 3: Model compilation time with Jacobians.

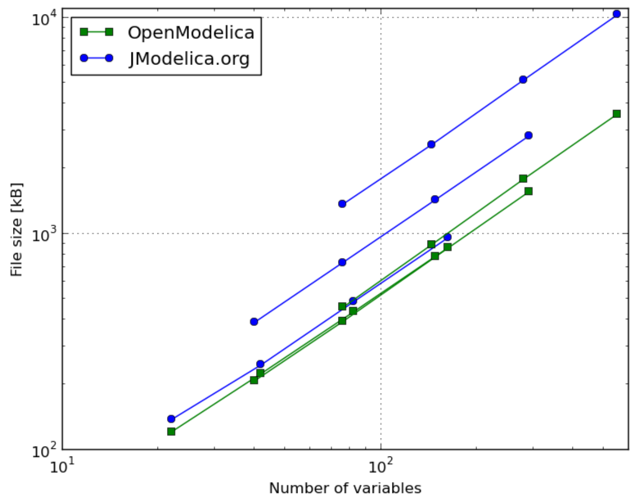

Figure 4: Generated code size with Jacobian.

5 Conclusions

In this paper, the generation of Jacobians for ODEs originating from DAEs, in particular Modelica models, has been discussed. The algorithmic machinery employed consists of known methods and algorithms, such as numerical, symbolic, and automatic differen-

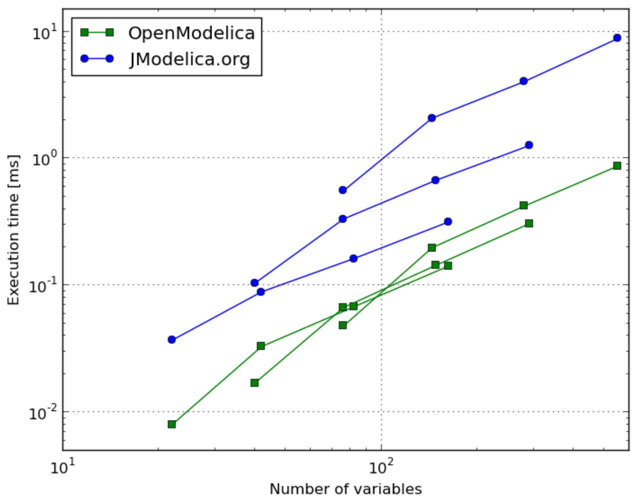

Figure 5: Execution time for one Jacobian evaluation

tiation, as well as graph theoretic methods such as the BLT transformation. Two methods, sharing similarities as well as differences have been presented. One of the methods is a straight forward application of forward automatic differentiation and generation of C code, which results in functions for the evaluation of directional derivatives, which in turn are used to compute Jacobians. The other method relies mainly on symbolic differentiation and makes use of symbolic simplification algorithms in a Modelica compiler to generate directional derivative functions. Both methods provide sparsity patterns for the ODE Jacobians, and they both make efficient use of sparsity in order to reduce the number of directional derivative evaluations, a technique referred to as compression.

The two approaches are implemented in JModelica.org and OpenModelica, respectively, and compared in an industrial benchmark as well as in several synthetic benchmarks. Both implementations show linear growth in key measures such as model compilation time, generated code size and execution time, under realistic assumptions on model structure. In terms of execution speed, the method relying on symbolic differentiation and symbolic processing, as implemented in OpenModelica, performed faster.

Memory consumption in the model compilation step was not included in the benchmarks, because of the inherent difficulties in accurately measuring this quantity. Indeed, this measure would have been an interesting addition to the benchmarks presented in this paper, especially since the two methods take different approaches to generate directional derivatives. However, measurements of memory consumption is left for future work.

6 Acknowledgments

Modelon's contribution to this work was partially funded by Vinnova within the ITEA2 project OPENPROD (dnr: 2010-00068). University of Applied Sciences Bielefeld's contribution to this work was partially funded by The German Ministry BMBF (01IS09029C) within the ITEA2 project OPENPROD. Johan Åkesson acknowledges financial support from Lund Center for Control of Complex systems, funded by the Swedish research council. The authors also would like to acknowledge the kind support from Francesco Casella, who provided the power plant model used for benchmarks.

References

[1] A. Griewank A. Walther. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition. SIAM, 2008.

[2] The Functional Mock-up Interface for Model Exchange 1.0, http://functional-mockup-interface.org/specifications/FMI_for_ModelExchange_v1.0.pdf, January 2010.

[3] The Functional Mock-up Interface for Co-simulation 1.0, http://functional-mockup-interface.org/specifications/FMI_for_CoSimulation_v1.0.pdf, October 2010.

[4] R. Tarjan. "Depth-first search and linear graph algorithms." SIAM J. Computing, **1:2**, pp. 146–160, 1972.

[5] D. H. Al-Omari K. E. Sabri. "New graph coloring algorithms." American Journal of Mathematics and Statistics, 2006.

[6] T. F. Coleman J. J. More. "Estimation of sparse Jacobian matrices and graph coloring problems." Society for Industrial and Applied Mathematics, 1983.

[7] Dürrbaum A., Klier W., Hahn H.: Comparison of Automatic and Symbolic Differentiation in Mathematical Modeling and Computer Simulation of Rigid-Body Systems. In: Multibody System Dynamics. Springer Netherlands, 2002.

[8] Y. B. Gol'dshtein. "Portrait of the inverse of a sparse matrix." Cybernetics and Systems Analysis, **28**, pp. 514–519, 1992.

[9] Braun W, Gallardo Yances S, Link K, Bachmann B. Fast Simulation of Fluid Models with Colored Jacobians . In: Proceedings of the 9th Modelica Conference, Munich, Germany, Modelica Association, 2012.

[10] Braun W, Ochel L, Bachmann B. Symbolically Derived Jacobians Using Automatic Differentiation - Enhancement of the OpenModelica Compiler. In: Proceedings of the 8th Modelica Conference, Dresden, Germany, Modelica Association, 2011.

[11] Casella, F., Donida, D., Åkesson, J. Object-Oriented Modeling and Optimal Control: A Case Study in Power Plant Start-Up. In: Proceedings of the 18th IFAC World Congress, Milan, Italy, 2011.