Symbolically Derived Jacobians Using Automatic Differentiation - Enhancement of the OpenModelica Compiler

Willi Braun Lennart Ochel Bernhard Bachmann
Bielefeld University of Applied Sciences, Department of engineering and mathematics
Am Stadtholz 24, 33609 Bielefeld
{wbraun,lochel,bernhard.bachmann}@fh-bielefeld.de

Abstract

Jacobian matrices are used in a wide range of applications - from solving the original DAEs to sensitivity analysis. Using Automatic Differentiation the necessary partial derivatives can be provided efficiently within a Modelica-Tool. This paper describes the corresponding implementation work within the OpenModelica Compiler (OMC) to create a symbolic derivative module. This new OMC-feature generates symbolically partial derivatives in order to calculate Jacobian matrices with respect to different variables. Applications presented here, are the generation of linear models of non-linear Modelica models and the usage of the Jacobian matrix in DASSL for simulating a model.

Keywords: Symbolic Jacobian, Automatic Differentiation, Linearization, DASSL, OpenModelica

1 Introduction

In the process of modeling and simulation the usage of derivatives in many stages of this process is very common. The derivatives are useful for simulating a model as well as for the sensitivity analysis [1] or the optimization [4] of models. The derivatives can be calculated in different ways. There exist numerical methods like finite difference, or symbolical methods as in algebra systems. But there is another method containing characteristics of both of them: Automatic Differentiation (AD) is the better choice over other ways for computing derivatives. It is accurate like symbolic differentiation, since the results are not affected by any truncation errors. AD is originally a numerical method in contrast to numerical differentiation that evaluates the derivative of a function specified by sequence of assignments in a computer program. Since a Modelica program is written with symbolic expressions, AD can be used to calculate symbolically partial derivatives. In this work the OMC is enhanced to provide the symbolic derivatives for a Modelica model using AD. The new OMC feature is applicable in a versatile way. As first application it is used for the linearization of non-linear models. The linearization of a non-linear model needs the calculation of partial derivatives with respect to some specific variables of the Modelica model. The partial derivatives are organized in so-called Jacobian matrices. For the linear model it must be calculated four different Jacobian matrices so that the main task is the calculation of symbolic partial derivatives for the linearization. A further application of this new OMC capability, is the usage of the derivatives for simulating a Modelica model. The commonly used implicit integration method DASSL is providing an interface for the symbolic Jacobian matrix. This feature can be now used to speed-up the solving time in OMC.

The structure of this paper is as follows: First, methods from AD theory are shortly introduced in order to calculate the symbolic derivatives. Afterwards a short introduction of the relevant Modelica language features is presented and the mathematical representation of the corresponding Modelica models is described. With this implementation we are able to differentiate almost the complete Modelica language elements supported by OpenModelica. Finally, the generation of a linear model and the usage of the symbolic derivatives for simulating a Modelica model with DASSL are presented as applications.

2 Automatic Differentiation

The calculation of the symbolic derivatives of a Modelica model is possible by using automatic differentiation (AD) methods. AD is an efficient method to calculate the derivative value for an algorithmic function. This technique is based on the fact that the derivatives of a function can be calculated by repeatedly applying mathematical rules to all the sequential elementary operations of a coded function. The elementary operations can be differentiated by applying the basic derivation rules

$$\begin{aligned} \nabla(u \pm v) &= \nabla u \pm \nabla v \\ \nabla(uv) &= u\nabla v + v\nabla u \\ \nabla(\frac{u}{v}) &= \frac{(\nabla u - \frac{u}{v}\nabla v)}{v} \end{aligned}$$

for the arithmetic operations and the chain rule

$$\nabla\phi(u) = \dot{\phi}(u)\nabla u$$

for differentiable functions ϕ (e.g. such as the standard functions $\sin(x), \cos x, \ldots$) with known derivatives. This approach is referred to in literature as "forward" mode [9].

For example, a function given by the formula 1 can be decomposed in the elementary operations as in table 1.

$$f(x_1, x_2) = (x_1 * x_2 + \sin(x_1))(3 * {x_1}^2 + x_2) \quad (1)$$

The basic rules of differentiation can be applied to the decomposed arithmetic operations to obtain the partial derivative of the function. Thus, the final results are the values $t_9 = f(x_1, x_2)$ of the function and its partial derivatives $\nabla f = \nabla t_9 = [(t_2 + \cos(t_1))t_8 + 6t_1t_5, t_1t_8 + t_5]$.

Since AD is originally a numerical method, it is common to determine the values only. If these terms are replaced by the original expressions that are available inside a Modelica compiler the symbolic derivative formulas are obtained. This automatic differentiation method can be used analogically in order to calculate the partial derivatives to the optimized DAEs as they occur in Modelica. This is possible, because the calculation of partial derivatives is performed by consistently applying the chain rule and the basic differentiation rules as mentioned above.

3 Differentiate a Modelica Model

A Modelica model is typically translated to a basic mathematical representation in terms of a flat system of differential and algebraic equations before being able to simulate the model. This translation process elaborates on the internal model representation by performing analysis and type checking, inheritance and expansion of base classes, modifications and redeclarations, conversion of connect equations to basic equations, etc. The result of this analysis and translation process is a flat set of equations, including conditional equations as well as constants, variables, and function definitions. By the term flat is meant that the object-oriented structure has been broken down to a flat representation where no trace of the object hierarchy remains, apart from dot notation (e.g. Class.Subclass.variable) within names.

Flat Modelica DAEs could be represented mathematically by the equation:

$$0 = F(\underline{\dot{x}}(t), \underline{x}(t), \underline{u}(t), \underline{y}(t), \underline{p}, t) \quad (2)$$

Below the notations used in the equation above are summarized:

- $\underline{\dot{x}}(t)$ the differentiated vector of state variables of the model.
- $\underline{x}(t)$ the vector of state variables of the model, i.e., variables of type Real that also appear differentiated somewhere in the model.
- $\underline{u}(t)$ a vector of input variables, i.e., not dependent on other variables, of type Real.They also belong to the set of algebraic variables since they do not appear differentiated.
- $\underline{y}(t)$ a vector of Modelica variables of type Real which do not fall into any other category.
- $\underline{p}$ a vector containing the Modelica variables declared as parameter or constant i.e., variables without any time dependency.

This implicit equation is transformed to the explicit state-space representation by the so-called block-lower-triangular (BLT) transformation resulting in the optimized DAEs. This transformation is done by a matching and sorting algorithm which results in a sequence of assignments so that the variables can be solved sequentially [7]:

Operations	Differentiate(t_i, $\{x_1, x_2\}$)	∇f
$t_1 = x_1$	$\nabla t_1 = [1, 0]$	$[1, 0]$
$t_2 = x_2$	$\nabla t_2 = [0, 1]$	$[0, 1]$
$t_3 = t_1 t_2$	$\nabla t_3 = t_1 \nabla t_2 + \nabla t_1 t_2$	$[t_2, t_1]$
$t_4 = \sin(t_1)$	$\nabla t_4 = \cos(t_1) \nabla t_2$	$[\cos(t_1), 0]$
$t_5 = t_3 + t_4$	$\nabla t_5 = \nabla t_3 + \nabla t_4$	$[t_2 + \cos(t_1), t_1]$
$t_6 = t_1 * t_1$	$\nabla t_6 = 2 t_1 \nabla t_1$	$[2 * t_1, 0]$
$t_7 = 3 * t_6$	$\nabla t_7 = 3 \nabla t_6$	$[6 * t_1, 0]$
$t_8 = t_7 + t_2$	$\nabla t_8 = \nabla t_7 + \nabla t_2$	$[2 * t_1, 1]$
$t_9 = t_5 * t_8$	$\nabla t_9 = t_5 \nabla t_8 + \nabla t_5 t_8$	$[t_2 + \cos(t_1)) t_8 + 6 t_1 t_5, t_1 t_8 + t_5]$

Table 1: Decomposed function $f(x_1, x_2)$ to elementary operations and the partial derivatives.

$$0 = F(\dot{\underline{x}}(t), \underline{x}(t), \underline{u}(t), \underline{y}(t), \underline{p}, t)$$

$$0 = F(\underline{z}(t), \underline{x}(t), \underline{u}(t), \underline{p}, t), \quad \underline{z}(t) = \begin{pmatrix} \dot{\underline{x}}(t) \\ \underline{y}(t) \end{pmatrix}$$

$$\underline{z}(t) = \begin{pmatrix} \dot{\underline{x}}(t) \\ \underline{y}(t) \end{pmatrix} = g(\underline{x}(t), \underline{u}(t), \underline{p}, t)$$

$$\begin{pmatrix} \dot{\underline{x}}(t) \\ \underline{y}(t) \end{pmatrix} = \begin{pmatrix} h(\underline{x}(t), \underline{u}(t), \underline{p}, t) \\ k(\underline{x}(t), \underline{u}(t), \underline{p}, t) \end{pmatrix} \tag{3}$$

This sequence of assignments can immediately be used for calculating the partial derivatives symbolically by means of automatic differentiation. Thus the differentiation process is performed on such optimized DAEs. These DAEs are separated in two partitions, a state block and an algebraic block. The function h represents the state block and consists of all equations, which are necessary to determine the differentiated states. The function k represents the algebraic block, which contains all remaining equations.

Consider, for example, the following small differential-algebraic system:

$$\begin{aligned} f_1 :=&\quad \dot{x}_1 = &\quad a * x_1 \\ f_2 :=&\quad \dot{x}_2 = &\quad a * x_2 + \dot{x}_1 \\ f_3 :=&\quad a = &\quad \sin(x_1) + \cos(x_2) \end{aligned}$$

To calculate all necessary partial derivatives the system has to be sorted by the BLT-Transformation based on the adjacency matrix:

$$\begin{array}{c} \\ f_1 \\ f_2 \\ f_3 \end{array} \begin{array}{c} \begin{array}{ccc} \dot{x_1} & \dot{x_2} & a \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ f_3 \\ f_1 \\ f_2 \end{array} \begin{array}{c} \begin{array}{ccc} a & \dot{x_1} & \dot{x_2} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

To get all partial derivatives with respect to the states the whole system needs to be differentiated, which means every equation has to be differentiated with respect to the states. This requires the derivatives of all known variables with respect to the states. In a Modelica model it is assumed that the known variables are the states and the inputs. In this example only the states appear as known variables:

$$\begin{pmatrix} \frac{\partial x_1}{\partial x_1} = 1 & \frac{\partial x_2}{\partial x_1} = 0 \\ \frac{\partial x_1}{\partial x_2} = 0 & \frac{\partial x_2}{\partial x_2} = 1 \end{pmatrix}$$

The next step is to take the sorted equations and differentiate straight forward applying the rules described above. With all the resulting partial derivatives it is possible to organize the Jacobian matrix with respect to the states $\underline{x}$ in equation (4). In the following, Modelica language features are described which need to be handled by the automatic differentiation.

Equations

The differentiation of ordinary equations is straightforward. In the optimized DAEs the matching algorithm provides information about the variable which has to be solved for in each equation. Therefore, the equations are rearranged to a corresponding assignment and differentiated. This also works for equations including if-expressions, where each branch will be differentiated, respectively. Non-linear equations will result into equations depending linearly on the differentiated variables (see example in the Algebraic loop section).

$$A = \frac{\partial f}{\partial \underline{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial \underline{x_1}} & \frac{\partial f_2}{\partial \underline{x_2}} \end{pmatrix} = \begin{pmatrix} \cos(x_1)x_1 + a & -\sin(x_2)x_1 \\ \cos(x_1)x_2 + \cos(x_1)x_1 + a & -\sin(x_2)x_2 + a - \sin(x_2)x_1 \end{pmatrix} \quad (4)$$

Algebraic loops

In many applications the transformation to the optimized DAEs cannot achieve a true lower-triangular form. It is at least possible to reduce the DAEs to a Block-Lower-triangular form with diagonal blocks of minimal size. These blocks are called algebraic loops and must be solved simultaneously. In general, this results in a system of linear and/or nonlinear equations.

For example the following equations have to be solved simulataneously:

$$\underline{f}(\underline{x}, \underline{p}, t) := \begin{pmatrix} \dot{x}_1 \\ \underline{\dot{x}_2} \end{pmatrix} = \begin{pmatrix} ax_1 + \frac{1}{2}\dot{x}_2{}^2 \\ bx_2 - \frac{1}{2}\underline{\dot{x}_1}{}^2 \end{pmatrix}$$

The equations are differentiated with respect to the state to determine the first row of the Jacobian matrix:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} := \frac{\partial \dot{x}_1}{\partial x_1} = a + \frac{\partial \dot{x}_2}{\partial x_1}\dot{x}_2 \\ \frac{\partial f_2}{\partial x_1} := \frac{\partial \dot{x}_2}{\partial x_1} = -\frac{\partial \dot{x}_1}{\partial x_1}\dot{x}_1 \end{pmatrix}$$

The resulting equation must still be solved simultaneously to determine the expressions for the first row of the Jacobian matrix. However, nonlinear equations that are differentiated, result always in equations depending linearly on the differentiated variables, which in this case, yield a linear system of equations to be solved.

Algorithms

Whereas equations are well suited to describe physical processes, there are situations where computations are more conveniently expressed by algorithms in a sequence of statements. In contrast to equations, statements are fixed assignments, i.e. the right-hand-side ones are assigned to the left-hand-side ones. Several assignments to the same variable can be performed in one algorithm section. Besides of simple assignment statements, an algorithm can contain if-, while-, and for-clauses. The symbolic differentiation can handle all of them.

Functions in Modelica

In Modelica, there exist two different types of functions, a Modelica function, written in Modelica code, and external functions that are written in C/Fortran code. A Modelica function is defined by an algorithm section that can be differentiated in the same way as algorithms. From the result a new Modelica function as the derivative to the original one is generated. This derivative function can be propagated by the derivative annotation to other process that needs the derivative. For external functions the numerical finite difference method is used, if that functions do not provide partial derivatives with the aid of the derivative annotation.

4 Applications for Symbolic Jacobian

4.1 Linear Models

A general nonlinear Modelica model is represented by state-space equations with n state variables, m input variables and k output variables:

$$\begin{pmatrix} \underline{\dot{x}}(t) \\ \underline{y}(t) \end{pmatrix} = \begin{pmatrix} h(\underline{x}(t), \underline{u}(t), \underline{p}, t) \\ k(\underline{x}(t), \underline{u}(t), \underline{p}, t) \end{pmatrix}$$

Linearizing the state-space equations the Taylor series expansion is applied and leads to a continuous-time linear dynamical system that has the form:

$$\underline{\dot{x}}(t) = A(t) * \underline{x}(t) + B(t) * \underline{u}(t)$$
$$\underline{y}(t) = C(t) * \underline{x}(t) + D(t) * \underline{u}(t)$$

$$A(t) = \frac{\partial h}{\partial \underline{x}} \in \mathbb{R}^{n \times n}, B(t) = \frac{\partial h}{\partial \underline{u}} \in \mathbb{R}^{n \times m}$$
$$C(t) = \frac{\partial \underline{k}}{\partial \underline{x}} \in \mathbb{R}^{k \times n}, D(t) = \frac{\partial \underline{k}}{\partial \underline{u}} \in \mathbb{R}^{k \times m}$$

The matrices $A(t)$, $B(t)$, $C(t)$, and $D(t)$ are the Jacobian matrices of the non-linear Modelica model. Thus the finding of linearization of a model is done by the calculation of the Jacobian matrices at a convenient time.

After all, the linear model can easily be generated when it's possible to differentiate a set of equations with respect to a set of variables. Therefore functions are implemented, that apply the method of forward automatic differentiation to given sets of equations, algorithms and variables. This function can deal with single equations, with

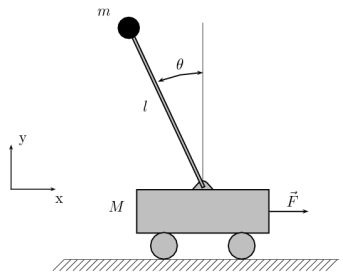

Figure 1: Schematic figure of inverse pendulum model.

systems of equations as well as algorithm sections and generate symbolically the needed Jacobian matrices for the linearization. The different steps of this procedure are sketched by the following model in Listing 1 and the corresponding schematic diagram of that model in Figure 1. In this model an inverse pendulum is balanced by a cart.

```
model InversePendulum
 parameter Real M = 0.5;
 parameter Real m = 0.2;
 parameter Real b = 0.1;
 parameter Real i = 0.006;
 parameter Real g = 9.8;
 parameter Real l = 0.3;
 parameter Real pi = 3.141592653589793;
 Real cart_x;
 Real cart_v;
 Real pendulum_theta;
 Real pendulum_w;
 output Real y[2];
 input Real u;
equation
  der(cart_x) = cart_v;
  der(pendulum_theta) = pendulum_w;
  (M + m)*der(cart_v) + b*cart_v +
  u = m*l*der(pendulum_w)*cos(pendulum_theta+pi)
  -m*l*pendulum_w^2*sin(pendulum_theta+pi);
  (i+m*l^2)*der(pendulum_w)+
  m*l*g*sin(pendulum_theta+pi)=
  -m*l*der(cart_v)*cos(pendulum_theta+pi);
  y={cart_x,pendulum_theta};
end InversePendulum;
```

Listing 1: InversePendulum model

The equations in Figure 2 are included into the generated simulation program and with this information the matrices A and C linearized model can be generated to any point in time. After compiling the generated C-code the evaluation of the linearized model at point in time 0 yields the Modelica model in Listing 2. The same simulation program can generate the linear model at any other point in time, i.e. equal 1, by simulating until then and afterwards evaluating the symbolic differentiated equations at this point.

```
model linear_InversePendulum
  parameter Integer n = 4; // states
  parameter Integer k = 1; // top-level inputs
  parameter Integer l = 2; // top-level outputs
  parameter Real x0[4] = {0,0,0,0};
  parameter Real u0[1] = {0};
  parameter Real A[4,4] =
  [0,1,0,0;
   0,-0.1818181818181819,2.672727272727272,0;
   0,0,0,1;
   0,-0.4545454545454546,31.18181818181818,0];
  parameter Real B[4,1] =
  [0;1.818181818181818;
  0;4.545454545454546];
  parameter Real C[2,4] = [1,0,0,0;0,0,1,0];
  parameter Real D[2,1] = [0;0];
  Real x[4](start=x0);
  output Real y[2];
  input Real u[1](start=u0);

  Real x_cart_x = x[1];
  Real x_cart_v = x[2];
  Real x_pendulum_phi = x[3];
  Real x_pendulum_w = x[4];
  Real u_u = u[1];
  Real y_y1 = y[1];
  Real y_y2 = y[2];

equation
  der(x) = A * x + B * u;
  y = C * x + D * u;
end linear_InversePendulum;
```

Listing 2: Linear Model of the InversePendulum at point in time 0

```
Equations (16)
========
1 : $DER$Pcart_x$pDERcart_x = 0.0
2 : $DER$Pcart_x$pDERcart_v = 1.0
3 : $DER$Pcart_x$pDERpendulum_theta = 0.0
4 : $DER$Pcart_x$pDERpendulum_w = 0.0
5 : $DER$Ppendulum_theta$pDERcart_x = 0.0
6 : $DER$Ppendulum_theta$pDERcart_v = 0.0
7 : $DER$Ppendulum_theta$pDERpendulum_theta = 0.0
8 : $DER$Ppendulum_theta$pDERpendulum_w = 1.0
9 : (M + m) * $DER$Pcart_v$pDERcart_x +
m * (l * ($DER$Ppendulum_w$pDERcart_x * cos(pendulum_theta + pi))) = 0.0
10 : (M + m) * $DER$Pcart_v$pDERcart_v +
(b + m * (l * ($DER$Ppendulum_w$pDERcart_v * cos(pendulum_theta + pi)))) = 0.0
11 : (M + m) * $DER$Pcart_v$pDERpendulum_theta +
m * (l * ($DER$Ppendulum_w$pDERpendulum_theta * cos(pendulum_theta +
pi) + (-der(pendulum_w)) * sin(pendulum_theta + pi))) -
m * (l * (pendulum_w ^ 2.0 * cos(pendulum_theta + pi))) = 0.0
12 : (M + m) * $DER$Pcart_v$pDERpendulum_w +
m * (l * ($DER$Ppendulum_w$pDERpendulum_w * cos(pendulum_theta + pi))) -
2.0 * (m * (l * (pendulum_w * sin(pendulum_theta + pi)))) = 0.0
13 : (i + m * l ^ 2.0) * $DER$Ppendulum_w$pDERcart_x =
(-m) * (l * ($DER$Pcart_v$pDERcart_x * cos(pendulum_theta + pi)))
14 : (i + m * l ^ 2.0) * $DER$Ppendulum_w$pDERcart_v =
(-m) * (l * ($DER$Pcart_v$pDERcart_v * cos(pendulum_theta + pi)))
15 : (i + m * l ^ 2.0) * $DER$Ppendulum_w$pDERpendulum_theta +
m * (l * (g * cos(pendulum_theta + pi))) =
(-m) * (l * ($DER$Pcart_v$pDERpendulum_theta * cos(pendulum_theta + pi)
+ (-der(cart_v)) * sin(pendulum_theta + pi)))
16 : (i + m * l ^ 2.0) * $DER$Ppendulum_w$pDERpendulum_w =
(-m) * (l * ($DER$Pcart_v$pDERpendulum_w * cos(pendulum_theta + pi)))
```

Figure 2: Equations for linear model matrices A and C

Such linear models are used in control theory for example as an observer to control the original nonlinear model [5].

4.2 Provide the analytical jacobian matrix to DASSL

For accurate, high-speed solution of DAEs as they occur in Modelica (see equation (2)) Petzold's Fortran-based DASSL(Differential-Algebraic System Solver) is the most widely used sequential code for solving such DAEs. After all, the DASSL implementation uses the following equation [8]

$$h(t, x, \hat{\alpha}x + \beta) = 0,$$

where $\hat{\alpha}$ is a constant which changes whenever the step size or the order changes, β is a vector which depends on the solution at past times and $t, x, \hat{\alpha}, \beta$ are evaluated at t_n. This equation is solved in DASSL by a modified version of Newton's method,

$$x^{m+1} = y^m - cj(\frac{\partial \underline{h}}{\partial \underline{x}} + cj * \frac{\partial \underline{h}}{\partial \underline{\dot{x}}})^{-1} h(t, x, \hat{\alpha}x + \beta).$$

The iteration matrix

$$M = \frac{\partial \underline{h}}{\partial \underline{x}} + cj * \frac{\partial \underline{h}}{\partial \underline{\dot{x}}}$$

is computed and factored, and is then used for as many time steps as possible.

By default DASSL calculates the iteration matrix M by the means of numerical finite differentiation. However, it is also possible to equip DASSL with an user-specific routine that provides the symbolically calculated iteration matrix M. On one hand, the symbolically calculated values are more accurate and on the other hand, it is faster to evaluate the symbolical formulas.

5 Conclusion and Future Work

The successful implementation of symbolically generated partial derivatives for the corresponding Jacobian matrices using automatic differentiation methods in the OpenModelica Compiler has been demonstrated. The new feature supports all Modelica language elements and Modelica models already handled by OMC. The corresponding symbolic derivative module has been validated by creating linear models for non-linear Modelica models. Futhermore, providing the analytically determined Jacobian matrix to DASSL, leads to a faster simulation of the model. In addition to this, this implemented methods offer a variety of different application fields (i.e. parameter identification, sensitivity analysis, uncertainty calculation, inline-integration methods, model reduction, optimization ...).

In future it is possible to improve this module in two directions: First this module could be made accessible for the user. The user could select some functions of equations (3) and some depended variables. Thus the user can decide which symbolic matrices is wanted in the simulation program. The second direction could be to generate directly a Modelica model with the symbolic derivative expressions. With this approach the symbolic matrices could be made accessible during simulation in a way so that the updated version can always be used for controlling or optimization processes.

Other initiatives aim to extend the Functional Mock-up Interface for model exchange [2] to support the evaluation of sparse Jacobians. This work can easily be adapted to provide the required calculations.

Acknowledgments

The German Ministry BMBF has partially funded this work (BMBF Förderkennzeichen: 01IS09029C) within the ITEA2 project OPENPROD (http://www.openprod.org).

References

[1] Elsheikh A., Noack S. and Wiechert W.: Sensitivity analysis of Modelica applications via automatic differentiation, 6th International Modelica Conference, Bielefeld, 2008.

[2] MODELISAR: Functional Mock-up Interface for Model Exchange, `http://modelisar.org/specifications/FMI_for_ModelExchange_v1.0.pdf`, Januar 2010.

[3] Fritzson P. et. al.: OpenModelica System Documentation, PELAB, Department of Computer and Information, Linköpings universitet, 2010.

[4] Imsland L., Kittilsen P. and Schei T.: Using Modelica models in the real time dynamic optimization - gradient computation, Proceedings 7th Modelica Conference, Como, 2009.

[5] Lunze, J.: Regelungstechnik 2 – Beobachterentwurf, Springer-Lehrbuch, Springer Berlin Heidelberg, 2010.

[6] Modelica Association: Modelica – A unified Object-oriented Language for Physical Systems Modeling Language Specification – Version 3.2, 2010.

[7] Otter M.: Objektorientierte Modellierung Physikalischer Systeme (Teil 4) Transformationsalgorithmen, Automatisierungstechnik, Oldenbourg Verlag München, 1999.

[8] Petzold L. R.: A Description of DASSL: A Differential/Algebraic System Solver, Sandia National Laboratories Livermore, 1982.

[9] Rall L.B.: Automatic differentiation: Techniques and applications, vol. 120 of Lecture Notes in Computer Science, Springer, 1981.