Towards Adjoint and Directional Derivatives in FMI utilizing ADOL-C within OpenModelica

Willi Braun[1] Kshitij Kulshreshtha[2] Rüdiger Franke[3] Andrea Walther[2] Bernhard Bachmann[1]

[1]FH Bielefeld University of Applied Science, {wbraun,bernhard.bachmann}@fh-bielefeld.de
[2]Universität Paderborn, kshitij@math.upb.de,andrea.walther@uni-paderborn.de
[3]ABB AG, Mannheim, ruediger.franke@de.abb.com

Abstract

Algorithmic differentiation has proven to be an efficient method for evaluating derivative information for implementations of mathematical functions. In the context of the Functional Mockup Interface (FMI) the reverse mode of algorithmic differentiation shows immense promise.

FMI is increasingly used for model-based applications, such as parameter estimation or optimal control. The paper motivates the exploitation of algorithmic differentiation and proposes an extension of FMI for the evaluation of adjoint directional derivatives.

Attempts to interface algorithmic differentiation libraries with Modelica tools have been made. Instead of generating code for the target language which is instrumented with algorithmic differentiation library API and then compiled, in this new approach the intermediate representation used by the library is generated directly. This avoids compilation of the target language that often takes a large fraction of the overall simulation time. It also avoids model execution in order to create such an internal representation at runtime. The initial results are presented here.

Keywords: OpenModelica, ADOL-C, Derivatives, Jacobian

1 Introduction

Algorithmic differentiation (Griewank and Walther, 2008) is a technique to compute derivatives of functions expressed as computer programs efficiently, and accurately upto machine precision (Griewank et al., 2012). Ruge et al. (2014) first investigated the use of the algorithmic differentiation tool ADOL-C (Walther and Griewank, 2012) in conjunction with OpenModelica (Fritzson et al., 2006). ADOL-C is designed for the C++ programming language and uses operator overloading to create an internal representation of the computation, called a trace, when a program instrumented with the datatype adouble is executed. In Ruge et al. (2014) such instrumented code written in C++ was generated in addition to the usual model code for the C-Runtime and was compiled and linked with the ADOL-C library in addition to the C-Runtime Library of OpenModelica.

In this work we endeavoured to generate the trace for the use by the ADOL-C library to compute derivatives directly from within the OpenModelica compiler. Since the compiler has all the required information about the computation of the model it can present this information in the manner we need, without having to generate C++ code and executing it. On the other hand the ADOL-C library did not have any other mechanism for creation the internal data structures associated with a trace, other than executing C++ code instrumented with the datatype adouble. The challenge was therefore two-fold: firstly to teach the ADOL-C library to accept a trace in another format, and secondly to generate this format from the OpenModelica compiler while processing the model.

This paper is organized as follows: Section 2 outlines the motivation for the exploitation of algorithmic differentiation in FMI. Further more it presents a proposal for an extension of FMI offering the evaluation of adjoint directional derivatives. The needed details of algorithmic differentiation as well as the implementation of ADOL-C are described in section 3. Whereas section 4 focuses on the implementation work in OpenModelica. Finally, the first results are shown in section 5.

2 Adjoint directional derivatives in FMI

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.

Starting from version 2.0, the Functional Mock-up Interface (FMI) is well suited to hook Modelica models to numerical solvers for model-based applications, such as parameter estimation or optimal control (Franke et al., 2015). This way numerical routines can focus on the solution process, while FMI abstracts implementation details of the model. It is likely that such applications of FMI will increase. FMI should provide appropriate model derivatives.

Consider the solution of a least squares problem for parameter estimation as example. A general model has the form

$$\begin{aligned} v_{unknown} &= h(v_{known}, v_{rest}), \\ & h : \mathbb{R}^{nKnown} \times \mathbb{R}^{nRest} \rightarrow \mathbb{R}^{nUnknown}. \end{aligned} \tag{1}$$

The task is to obtain *nKnown* parameters such that a sum of squared residuals for *nUnknown* model outputs y and $k = 1, \ldots, K$ given data points y_k is minimized:

$$R = \sum_{k=1}^{K} \| y_k - h(v_{known}, v_{rest,k}) \|^2 \rightarrow \min_{v_{known}} . \tag{2}$$

The solution must fulfill the necessary condition

$$\begin{aligned} \frac{\partial R}{\partial v_{known}} &= \text{zeros}(nKnown) = \\ -2 \sum_{k=1}^{K} & \left[y_k - h(v_{known}, v_{rest,k}) \right] \frac{\partial h(v_{known}, v_{rest,k})}{\partial v_{known}}. \end{aligned} \tag{3}$$

The solution process, for instance applying Newton's method, involves the successive computation of (3), including model derivatives.

The existing API of FMI 2.0 provides the function fmi2GetDirectionalDerivative to obtain a column of the Jacobian matrix $\partial h(v_{known}, v_{rest}) / \partial v_{known}$ or a linear combination of columns of the Jacobian matrix. One computation of directional derivatives gives:

$$\Delta v_{unknown} = \frac{\partial h(v_{known}, v_{rest})}{\partial v_{known}} \Delta v_{known}. \tag{4}$$

The signature of the API function is:

```
fmiStatus fmi2GetDirectionalDerivative(
  fmiComponent c,
  const fmi2ValueReference vUnknown_ref[],
  size_t nUnknown,
  const fmi2ValueReference vKnown_ref[],
  size_t nKnown,
  const fmi2Real dvKnown[],
  fmi2Real dvUnknown[])
```

The computation of (3) requires $K \times nKnown$ calls to fmi2GetDirectionalDerivative . This is inefficient if multiple parameters shall be estimated ($nKnown > 1$).

This is why the FMI interface should be extended with a new function fmi2GetAdjointDerivative, along with a capability flag providesAdjointDerivatives . The new function computes:

$$\Delta v_{known} = \left(\frac{\partial h(v_{known}, v_{rest})}{\partial v_{known}} \right)^T \Delta v_{unknown}. \tag{5}$$

It has the signature:

```
fmiStatus fmi2GetAdjointDerivative(
  fmi2Component c,
  const fmi2ValueReference vUnknown_ref[],
  size_t nUnknown,
  const fmi2ValueReference vKnown_ref[],
  size_t nKnown,
  const fmi2Real dvUnknown[],
  fmi2Real dvKnown[])
```

The new function allows to obtain one row of the Jacobian matrix, or a linear combination of rows of the Jacobian matrix, with only one model evaluation in reverse mode of algorithmic differentiation. The computation of (3) becomes significantly more efficient. Only K calls to fmi2GetAdjointDerivative are needed, one call for each data point k and arbitray numbers of *nKnown* parameters or *nUnknown* model outputs, when passing the values of the residuals as seeds

$$\Delta v_{unknown,k} = y_k - h(v_{known}, v_{rest,k}). \tag{6}$$

3 Algorithmic differentiation using ADOL-C

In order to apply algorithmic differentiation (AD) on a program we model the program structure as a sequence of instructions, which perform specific mathematical functions. This is called an evaluation procedure in Griewank and Walther (2008). The evaluation procedure can be then evaluated forwards or reverse to compute the derivatives in the so called forward mode and reverse mode of AD. Griewank and Walther (2008, Chapter 3 and 4) describe this process in great detail and give bounds on the compexity and memory requirements. The main import of the complexity analysis is that the reverse mode is very well suited to compute gradient vectors for functions in much less complexity than they can be computed otherwise, either numerically or symbolically. The same applies to computing rows of the Jacobian matrix.

ADOL-C implements the AD process by overloading the operators and mathematical functions in the C++ programming language for a special datatype adouble. Such supported operations are called elementary operations. Each of these overloaded operators and functions when executed records the elementary operation currently being performed, the locations of the operands in working memory, and the locations of the results in working memory. This record is created normally on runtime, when a program, instrumented with the ADOL-C headers, datatypes and some instructions on when to begin and end the recording, is executed. ADOL-C is then able to use this record, called a *trace*, to evaluate function values, first

DOI 10.3384/ecp17132363

and higher order derivatives in forward and reverse mode at any given point of evaluation. The trace can be made persistent even after the program has terminated. Traditionally this trace is stored in binary format as raw data in three different files, one for the list of operations, one for the list of operand locations, and one for storing any constant values that might occur. The organisation of the trace is as follows. For each operation a character to represent it is stored. Based on what the operation actually is, it will requires a number of operands, which are stored as unsigned integer locations inside a data buffer, followed by the location, where the result of the operation will be stored. In some operations a constant may be involved, this value is stored as is.

In the work of Ruge et al. (2014) the Modelica models were translated into C++ code instrumented with the ADOL-C headers and datatypes, using a mechanism similar to the generation of C code for the model in OpenModelica. The drawback was that compilation linking and one-time execution of this C++ code was quite slow due to the use of operator overloading, compared to the generated C code. It was suggested, that since OpenModelica was already analysing the model in great detail, could we not create the trace of the model directly instead of generating C++ code.

```
// define independent
{ op:assign_ind loc:0 }
{ op:assign_ind loc:1 }
// operations
{ op:mult_d_a loc:0 loc:4 val:-0.25 }
{ op:div_a_a loc:1 loc:0 loc:5 }
{ op:plus_a_a loc:4 loc:5 loc:6 }
{ op:plus_d_a loc:6 loc:3 val:3.0 }
{ op:log_op loc:0 loc:4 }
{ op:mult_d_a loc:4 loc:5 val:-3.0 }
{ op:plus_a_a loc:1 loc:5 loc:2 }
// define depenpendent
{ op:assign_dep loc:2 }
{ op:assign_dep loc:3 }
// death_not
{ op:death_not loc:0 loc:8 }
```

Figure 1. An example of a texual trace for ADOL-C

OpenModelica is able to generate textual information rather than binary. Therefore the first step required for creating a trace directly was to allow a textual representation of the trace and that ADOL-C understands such a textual representation. A simple ASCII representation of the operations, locations and constants was devised with some delimiters to make parsing easier. Each elementary operation supported by ADOL-C is given a textual name stored with the keyword `"op:"`. The locations of all the operands inside the work buffer in decimal notation follow this and then the location of the result in the work buffer, each of these using the keyword `"loc:"`. At the end any required constant for the particular operation is given in decimal floating point notation using the keyword `"val:"`. Braces separate one such record from another. This textual representation is a natural extension of the binary representation for ADOL-C traces, which has long been a part of the ADOL-C public API. An ADOL-C driver function can now be used to convert any traditional binary trace to this textual representation and store it in a file. This format will be made a part of the public API of ADOL-C in the next feature release. An example of a file containing such a textual trace is shown in Figure 1.

A driver was added to ADOL-C to be able to read and parse a text file in ASCII notation with the above information using regular expressions to match the format described above and convert it to the traditional binary notation at runtime. Anything not matching the defined regular expressions is considered a comment and ignored.

4 Generation of Operation Lists

In the first step of the compilation process in Modelica tool, a model is transformed by the front-end into a flat representation, consisting essentially of lists of variables, functions, equations and algorithms. In this phase, a basic structural analysis of the differential-algebraic equations (DAE) is performed to detect the states and discrete variables and eliminate alias variables. The basic step of a Modelica compiler is to causalize the DAE and transform into ordinary differential equations (ODE). Then the target code is generated from the optimized system in order to perform the simulation. For the simulation the generated code needs to be compiled by the target language compiler and linked with the simulation runtime library. The default target language of the OpenModelica Compiler (OMC) is C and the GNU C compiler is used as default C compiler. In order to generate operation lists by OMC, which are readable by ADOL-C, the code generation module of OMC has been extended by a new target, the ADOL-C target. Basically the operation lists are generated by traversing the equation expressions and for every mathematical operation creating the corresponding ADOL-C operation. This is straight forward for assignments thus the OMC has transformed all equations into assignments due to the causalization. The implicit equations of the strong connected components require special treatment (Griewank and Walther, 2008). Furthermore, the OpenModelica simulation runtime is linked with the ADOL-C library in order to enable the usage of the ADOL-C capability during the simulation process. At the current status of the implementation we evaluate the sparse Jacobian for the integration process. One main advantage over the former approach is that the compilation of the target language can be avoided by processing the operation lists directly.

5 First Results

The first results to test the performance of the approach presented here are based on benchmark models from the ScalableTestSuite library (Casella, 2015). In the current implementation status the sparse jacobian evaluation used

by the time integration method ida is used for evaluation. In the tables 1 and 2 all numbers are produced by using the model ScalableTestSuite.Elementary.SimpleODE.Models.CascadedFirstOrder.

Table 1. Evaluation time of the Jacobian. Compare OMC symbolic vs. ADOL-C

N	ADOL-C	OM Symbolic
100	0.000480442	0.000156783
200	0.000830835	0.000413299
400	0.00157551	0.000952923
800	0.00294508	0.00209405
1600	0.00676732	0.00536921
3200	0.0141433	0.012003
6400	0.0390204	0.0310391
12800	0.0771545	0.0756394
25600	0.1532143	0.1621433

In table 1 the time of one Jacobian evaluation is stated, calculated by $\frac{totalTime}{N}$, where $totalTime$ is the time that is needed to evaluate the Jacobian over the entire simulation horizon and N is the number of evaluations done. One can see that the evaluation time for the given model is quite equal between the generated symbolic Jacobian by OMC and the evaluation by ADOL-C. Note that ADOL-C is performing additional work (e.g. memory allocation and colouring) in the first call.

Table 2. Generation performance of Jacobian. Compare OMC symbolic vs. ADOL-C

	ADOL-C		OM Symbolic	
N	generate	read	generate	compile
100	0.00046	0.01475	0.015	
200	0.00089	0.02879	0.032	
400	0.00178	0.05794	0.059	
800	0.00372	0.11320	0.119	0.03
1600	0.00860	0.22766	0.244	0.14
3200	0.01749	0.45620	0.523	0.38
6400	0.03702	0.91150	1.229	0.48
12800	0.07571	1.82352	2.569	1.01
25600	0.15910	3.60362	5.459	1.65

The performance of generating the appropriate Jacobian is stated in table 2. These timings are divided in two stages. For ADOL-C it is time for the generation of the operation list, and the time to read them at runtime. For the symbolic Jacobian generated by OMC it is the generation of directional derivative code and the additional time to compile the generated C code. This result shows the linear complexity of the new approach presented in this paper.

6 Conclusion and Future work

This paper presents a new approach to generate a model evaluation trace for algorithmic differentiation, where no compilation of the model code is needed any more. The advantage of this approach is not only good performance, moreover it gives access to a feature-rich AD tool (e.g. higher-derivative, reverse mode). Furthermore, an extension of FMI involving adjoint derivatives is proposed and motivated by optimization-based applications, where such derivatives are mandatory. The implementation of this extension can be achieved by the approach described here. However, this requires some more implementation work, since the current implementation does not yet support all Modelica language features. The most important and challenging aspect is the treatment of implicit equations. In future the authors will continue working on supporting more language features with the approach described. Further, the here proposed FMI extension will be implemented and demonstrated with a complex example.

7 Acknowledgments

The presented work is part of the PARADOM project, that is funded by the Federal Ministry of Education and Research (BMBF) under the support code 01IH15002.

References

F. Casella. Simulation of large-scale models in Modelica: State of the art and future perspectives. In P. Fritzson and H. Elmqvist, editors, *Proceedings* 11^{th} *International Modelica Conference*, pages 459–468, Versailles, France, Sep 21–23 2015. The Modelica Association. doi:10.3384/ecp15118459.

R. Franke, M. Walther, N. Worschech, W. Braun, and B. Bachmann. Model-based control with FMI and a C++ runtime for Modelica. In *Proceedings of the 11th International Modelica Conference*. Modelica Association, Paris, France, Sep. 2015.

P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli, and D. Broman. Openmodelica - a free open-source environment for system modeling, simulation, and teaching. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1588 –1595, oct. 2006. doi:10.1109/CACSD-CCA-ISIC.2006.4776878.

A. Griewank and A. Walther. *Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM, 2008.

A. Griewank, K. Kulshreshtha, and A. Walther. On the numerical stability of algorithmic differentiation. *Computing*, 94 (2-4):125–149, 2012.

V. Ruge, W. Braun, B. Bachmann, A. Walther, and K. Kulshreshtha. Efficient implementation of collocation methods for optimization using openmodelica and ADOL-C. In H. Tummescheit and K.-E. Årzén, editors, *Proceedings of the* 10^{th} *Modelica Conference*, pages 1017–1025, Lund, Sweden, 2014. Modelica Assiciation and Lund University Electronic Press. doi:10.3384/ECP140961017.

A. Walther and A. Griewank. Getting started with ADOL-C. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, pages 181–202. Chapman-Hall, 2012.