Operational Semantics for a Modular Equation Language

Christoph Höger

Department of Software Engineering and Theoretical Computer Science, Technische Universität Berlin, Germany,
christoph.hoeger@tu-berlin.de

Abstract

Current equation–based object–oriented modeling languages offer great means for composition of models and source code reuse. Composition is limited to the source level, though: There is currently no way to compose pre-compiled model fragments. In this work we present $t^{(n,p)}$, a language which aims to overcome this deficiency. By using automatic differentiation directly in the language semantics, $t^{(n,p)}$offers the ability to implement index-reduction and causalisation of equation-terms without knowing their source-level representation. The semantics of $t^{(n,p)}$allow for calculation of arbitrary-order partial and total derivatives of pre-compiled terms.

Keywords Composition, Equation, DAE, Separate Compilation, Automatic Differentiation

1. Introduction

Equation based modeling languages offer great means for model composition and reuse. Unfortunately, this feature vanishes during classical model instantiation. There is no such thing as a Functional Mockup Unit ([1], a standard for composing compiled ordinary differential equations) for a differential algebraic equation (DAE).

The reason for this seems to be an implementation detail in the most common implementations: Usually e.g. a Modelica implementation will *interpret* the global model once and afterwards *compile* the resulting DAE into efficient executable code.

This "one-time instantiation" approach has some downsides:

- It favors a whole-model approach for static analysis: As there is no necessity for any interface abstraction, errors caused by wrong composition are only detected just prior to simulation.
- The code generation tends to *scale* badly in the case of large, uniform models [5, 13], because the symbolic manipulation of every equation leads to huge amounts of generated code.
- It prevents highly dynamic structural models: If a model *computes* (i.e. by a Turing-complete language) its succeeding mode, it is in general impossible to predict all modes of operation.

One way to overcome all those limitations is to find a way to compile equations *separately* into a form that can be instantiated arbitrarily often. Yet the attempt to do so reveals that the one-time approach is not just an implementation detail, since simulation often depends on the manipulation of the global system of equations. Thus, separate compilation needs to preserve the possibility to apply these manipulations.

As we will show, the main operation of index-reduction –differentiation of terms– can be implemented by automatic differentiation. Additionally, our method allows for sorting the compiled equations and solving them efficiently.

The rest of this paper is organized as follows: First we motivate the need for arbitrary-order differentiation and parameter selection of equations in a compiled setting. Afterwards we formally define $t^{(n,p)}$, a family of languages that allows for arbitrary order differentiation of mathematical terms, which we have implemented in a a general purpose DAE library called jdae[1]. We prove that the evaluation of this language indeed yields correct partial and total derivatives of a function. Finally, we show how $t^{(n,p)}$ can be implemented recursion-free.

1.1 Notation

In this paper we will use some short cuts to enhance the readability of formulas:

First, as we are talking about continuous functions, we will usually name them with the letters f, g, h. Domain and codomain are written in a blackboard style, i.e. $f : \mathbb{R} \to \mathbb{R}$ means that f is a function mapping real numbers to real numbers. Arguments are usually vectors ($\bar{x}$) or scalars (v). When the context is clear, we extend primitive operations over continuous functions: $(f + g)(v, \bar{x}) \equiv f(v, \bar{x}) + g(v, \bar{x})$. For function application we use a "flat" format: If $f : \mathbb{R}^{n+1} \to \mathbb{R}$, $v \in \mathbb{R}$ and $\bar{x} \in \mathbb{R}^n$, then we write $f(v, \bar{x})$ to denote the application of f to the concatenation of v to $\bar{x}$.

[1] https://github.com/choeger/jdae

The 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop
December 3, 2013, Vancouver, Canada.

 Proceedings are available at:
http://dx.doi.org/10.3384/ecp13090

AVICPS website:
http://www.analyticintegration.org/

2. Motivation

To understand the two most important operations on systems of differential and algebraic equations (namely index-reduction and causalisation), we resort to the classic higher index example, the cartesian pendulum:

$$x^2 + y^2 = 1 \quad (1)$$
$$\ddot{x} = Fx \quad (2)$$
$$\ddot{y} = Fy - 9.81 \quad (3)$$

As always, x and y denote the pendulum's coordinates while F is the tension force. In this example, we set the length of the pendulum and its mass to 1 to enhance readability.

2.1 Index Reduction

It is well known, that the above model cannot be solved directly by an ODE solver. The reason is obviously that both x and y appear differentiated but only one of them can be solved by equations 2 or 3, as one these equations is required to solve for F.

The solution to this problem is naturally to differentiate equation 1. By application of simple arithmetic laws the above model also implies:

$$\dot{x}x + \dot{y}y = 0 \quad (4)$$
$$\dot{x}^2 + \ddot{x}x + \dot{y}^2 + \ddot{y}y = 0 \quad (5)$$

As one can easily see, the augmented system of equations can be solved as an ODE (by choosing either x or y as state).

This result can be obtained by using an index-reduction algorithm like e.g. Pantelides' method [10], the dummy derivative method [8] or Pryce' method [11]. Any such method will result in the number of times a given equation or variable needs to be differentiated (explicitly as vectors d and c in Pryce' algorithm). Thus a compiled equation needs to be able to, given an arbitrary n, compute the n-th total derivative of itself.

2.2 Causalisation

In addition to the total derivative, it is usually necessary to compute the partial derivatives of an equation. This is due to the fact that most models will require to solve some algebraic parts iteratively. To do so efficiently, it is a common requirement to calculate the Jacobian of the equations.

In our example (assuming we choose x as state), we might solve y and $\dot{y}$ directly by equations 1 and 4 (as x and $\dot{x}$ are known by numerical integration), but finding a valid solution for F, $\ddot{x}$ and $\ddot{y}$ requires the iterative solution of equations 2, 3 and 5.

The process of finding such a partitioning of the system is called causalisation or equation-sorting. Given an index-1 DAE, the process is equivalent to the contraction of all strongly connected components in the dependency graph of equations.

This process raises an interesting problem in the setting of compiled equations: As the causalisation is applied after compile time, it is unknown for a given equation, which of the occurring variables are true iteration variables (e.g. F in the algebraic loop above) during simulation and which become constants (e.g. y).

2.3 Automatic Differentiation

In summary, a compiled equation needs to be able to compute for any order of total derivation and any subset of its variables the value and the partial derivatives of its residual. As we do not know neither the subset of variables nor the final degree of derivation, the only practical solution to this requirement is automatic (or algorithmic) differentiation. This technique is based on the fact that the derivation rules for primitive operations like addition or sine are well known (up to an arbitrary degree) and the chain rule shows how any composition of those primitive operations can be derived.

In this work we present a novel approach that embeds AD into a small term-language and prove its correctness. Our approach is hand-tailored to compute exactly the derivatives needed by a DAE solver. For any further reading about automatic differentiation we refer to section 7.3.

3. The term language t

In this section we define the simple term language t. We define language syntax in form of an EBNF-grammar. t is defined as:

$$\begin{array}{rcl} \tau & ::= & \mathtt{u}_{n,d} \\ & | & \tau \oplus \tau \\ & | & \tau \otimes \tau \\ & | & \phi\, \tau \\ & | & r \in \mathbb{R} \end{array}$$

t is a rather simple language: Terms (in the following sections abbreviated by variables τ_i) are either multiplication (written $\otimes$ to distinguish multiplication-terms from multiplication on real numbers which we will write as $\times$ further down), addition (written as $\oplus$), real values (r) and primitive functions $\phi \in \mathtt{P}$, where $\mathtt{P}$ is a not further defined set of n-times continuously differentiable (i.e. of class C^n) single-argument real-valued functions. Terms describing the application of a primitive function ϕ on a term τ are written as juxtaposition $\phi\, \tau$ while parentheses (i.e. $\phi(r)$ indicate the result of the actual application on real-numbers.

A notable feature of t is the availability of unknowns $\mathtt{u}_{n,d}$. Informally an unknown, written by $\mathtt{u}$ subscripted by two natural numbers, e.g. $\mathtt{u}_{n,d}$, represents the d-th total derivative of the n-th variable of a system of equations.

3.1 Semantics

The operational semantics of t is given below in the form of natural semantics. For an open interval $\mathbb{D} \subseteq \mathbb{R}$, a vector $\bar{x} = (x_1, \ldots, x_p) \in (\mathbb{D} \to \mathbb{R})^p$ of functions of class C^n and a free variable $v \in \mathbb{D}$, the evaluation relation $\Downarrow$ is defined. We write $(\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow r$ to denote that under $(\mathbb{D}, \bar{x}, v)$ τ evaluates to r. Intuitively, $(\mathbb{D}, \bar{x}, v)$ denotes the domain of a ideal residual computation: $\bar{x}$ is the vector of unknowns, v is the independent variable and $\mathbb{D}$ is the

interval upon which the model is well-posed (i.e. n-times continuously differentiable).

$\Downarrow$ is defined inductively by rules in form of a natural (or big-step) semantics. Each rule contains zero or more premises (the part above the bar) and one conclusion (the part below). The whole rule forms an implication: If the premises are true, the same holds for the conclusion. Multiple premises form an implicit conjunction. We also write side-conditions as part of the premises to save some space.

$$\frac{}{(\mathbb{D}, \bar{x}, v) \vdash r \Downarrow r} \quad \text{(REAL)}$$

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow r_1 \quad (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \Downarrow r_2}{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \oplus \tau_2 \Downarrow r_1 + r_2} \quad \text{(ADD)}$$

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow r_1 \quad (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \Downarrow r_2}{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \otimes \tau_2 \Downarrow r_1 \times r_2} \quad \text{(MUL)}$$

$$\frac{r \in \mathbf{dom}(\phi) \qquad (\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow r}{(\mathbb{D}, \bar{x}, v) \vdash \phi\, \tau \Downarrow \phi(r)} \quad \text{(PRIM)}$$

$$\frac{n \in 0 \ldots p}{(\mathbb{D}, \bar{x}, v) \vdash \mathtt{u}_{n,d} \Downarrow x_n^{(d)}(v)} \quad \text{(UNK)}$$

Definition 1. *Let $f : (\mathbb{D} \to \mathbb{R})^p \times \mathbb{D} \to \mathbb{R}$ be an n-times differentiable function, then:*

$$(\mathbb{D}, \bar{x}, v) \vdash \tau \rightsquigarrow f :\Leftrightarrow (\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow f(\bar{x}, v)$$

(τ computes f)

To conclude this section, we state an observation about the well-formedness of terms of t. We call a term well formed (under $(\bar{x}, v)$) if there is an $r \in \mathbb{R}$ such that $(\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow r$. It should be obvious that the only form of non-reducing terms can occur due to E-PRIM-REAL or E-UNK. For the proofs further below, we eliminate this possibility by the following lemma (which could be easily shown, since t is a strict language):

Lemma 1. *Well formed terms always compute a function and terms that compute a function are well-formed.*

4. The $t^{(n,p)}$ Family

To implement automatic differentiation, we lift the simple language t into a family of languages $t^{(n,p)}$.

$t^{(n,p)}$ is syntactically very similar to t. Again, we see addition, multiplication, primitive functions and unknowns:

$$\begin{aligned} \tau^{(n,p)} \quad ::= \quad & \mathtt{u}_{m,d} \\ \mid \quad & \tau^{(n,p)} \oplus \tau^{(n,p)} \\ \mid \quad & \tau^{(n,p)} \otimes \tau^{(n,p)} \\ \mid \quad & \phi\, \tau^{(n,p)} \\ \mid \quad & \mathtt{A} \in \mathbb{R}^{(n+1,p+1)} \end{aligned}$$

The only visible difference to t is that the domain of real values is exchanged with a domain of real-valued matrices (which we abbreviate with capital latin letters). ϕ still denotes real-valued functions. The intuitive explanation is that a $t^{(n,p)}$-term calculates not only a value, but also the n total derivatives and the p partial derivatives (of every total derivative) of a function.

Two terms of different instances of $t^{(n,p)}$can only differ in the shape of the constants. It is important to note that if a term does not contain any constants, it is a valid $t^{(n,p)}$-term for any given concrete n and p.

4.1 Fundamental Definitions

Before we can explain the semantics of $t^{(n,p)}$, we need to introduce some helper functions. First, we define a lifting operator $\lceil\rceil_p^n$ that allows us to map a t-term into a $t^{(n,p)}$-term:

$$\begin{aligned} \lceil r \rceil^{(n,p)} &= \begin{vmatrix} r & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{vmatrix} \\ \lceil \tau_1 \oplus \tau_2 \rceil^{(n,p)} &= \lceil \tau_1 \rceil^{(n,p)} \oplus \lceil \tau_2 \rceil^{(n,p)} \\ \lceil \tau_1 \otimes \tau_2 \rceil^{(n,p)} &= \lceil \tau_1 \rceil^{(n,p)} \otimes \lceil \tau_2 \rceil^{(n,p)} \\ \lceil \mathtt{u}_{m,d} \rceil^{(n,p)} &= \mathtt{u}_{m,d} \\ \lceil \phi\, \tau \rceil^{(n,p)} &= \phi\, \lceil \tau \rceil^{(n,p)} \end{aligned}$$

Additionally we introduce two reduction operations on real-matrices called Δ (for differentiation) and I (for integration). The naming will be obvious once we define the operational semantics of $t^{(n,p)}$, but for now they are defined by elimination of the first and last row respectively:

$$I \begin{vmatrix} r_{0,0} & \cdots & r_{0,p} \\ \vdots & \ddots & \vdots \\ r_{n,0} & \cdots & r_{n,p} \end{vmatrix} = \begin{vmatrix} r_{0,0} & \cdots & r_{0,p} \\ \vdots & \ddots & \vdots \\ r_{n-1,0} & \cdots & r_{n-1,p} \end{vmatrix}$$

$$\Delta \begin{vmatrix} r_{0,0} & \cdots & r_{0,p} \\ \vdots & \ddots & \vdots \\ r_{n,0} & \cdots & r_{n,p} \end{vmatrix} = \begin{vmatrix} r_{1,0} & \cdots & r_{1,p} \\ \vdots & \ddots & \vdots \\ r_{n,0} & \cdots & r_{n,p} \end{vmatrix}$$

For the definition of the semantics as well as the proof of correctness, we will require a notion of mixed total and partial derivation. To reduce syntactical noise we will stick with the well-known d and ∂ notations, but omit any fractions:

$$\begin{aligned} \partial_0 f &= f \\ \partial_j f &= \frac{\partial f}{\partial x_j}, \quad i \in 1 \ldots p \\ \mathtt{d}^{(i)} f &= \frac{\mathtt{d}^i f}{\mathtt{d} v^i} \end{aligned}$$

This notation ignores the partial derivative $\frac{\partial f}{\partial x_0}$. The reason for this decision is the matrix layout used further down: It is convenient to use the same index-set for the delta operator as in the matrix. To reconcile this restriction with our

general mathematical model, we introduce the convention that $x_0(v) = v$, thus the missing partial derivative is identical to the first total derivative.

Matrix values will be abbreviated with capital latin letters, so $A \in \mathbb{R}^{n+1,p+1}$ in the context of $t^{(n,p)}$. When we index a matrix with one number, we select the corresponding row-vector, i.e. A_0 is the vector containing the elements of the first row of A.

To store partial and total derivatives, we introduce the n, p-dimensional derivative-matrix $\mathcal{D}^{(n,p)}(f, \bar{x}, v)$ of a function f.

$$\mathcal{D}^{(n,p)}(f, \bar{x}, v) \in \mathbb{R}^{n+1,p+1}$$
$$\mathcal{D}^{(n,p)}(f, \bar{x}, v)_{(i,j)} = \partial_j \mathtt{d}^{(i)} f(\bar{x}, v)$$

In case the context is clear, we omit the $\bar{x}, v$ arguments for brevity.

This definition enables the formulation of a correctness theorem on the evaluation of $t^{(n,p)}$-terms. It needs to be based on the fundamental assumption that a term actually computes a function. If this was not the case, the output of the reduction might still yield values but they are hardly meaningful in the sense of derivation.

Theorem 1 (correctness)**.**

$$(\mathbb{D}, \bar{x}, v) \vdash \tau \rightsquigarrow f \Rightarrow (\mathbb{D}, \bar{x}, v) \vdash \lceil \tau \rceil_p^n \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(f)$$

(the result of computing a function is the matrix of its derivatives)

The formal definition of $\Downarrow^{(n,p)}$ (the evaluation semantics of $t^{(n,p)}$as a binary relation between $t^{(n,p)}$-terms) follows below. As we will show, it fulfills this theorem.

We also introduce a special vector operator $\star : \mathbb{R}^{p+1} \times \mathbb{R}^{p+1} \to \mathbb{R}^{p+1}$, which is defined as follows:

$$(\bar{a} \star \bar{b})_0 = a_0 \times b_0$$
$$(\bar{a} \star \bar{b})_i = a_i \times b_0 + b_i \times a_0 \qquad i \in 1 \dots p$$

The motivation of this definition lies in the following fact:

Lemma 2. *Let $g, h : (\mathbb{D} \to \mathbb{R})^p \times \mathbb{D} \to \mathbb{R}$ be differentiable functions with p parameters each, then:*

$$\mathcal{D}^{(0,p)}(g, \bar{x}, v) \star \mathcal{D}^{(0,p)}(h, \bar{x}, v) = \mathcal{D}^{(0,p)}(g \times h, \bar{x}, v)$$

A second operator, $\bullet : ((\mathbb{R} \to \mathbb{R}) \times \mathbb{R}^{p+1}) \to \mathbb{R}^{p+1}$, takes a differentiable function and a $p+1$ vector and returns a $p+1$ vector:

$$(\phi \bullet \bar{a})_0 = \phi(a_0)$$
$$(\phi \bullet \bar{a})_i = \phi'(a_0) \times a_i \qquad i \in 1 \dots p$$

This composition operator is also motivated by an associated corollary:

Lemma 3. *Let $\phi : \mathbb{R} \to \mathbb{R}, g : (\mathbb{D} \to \mathbb{R})^p \times \mathbb{D} \to \mathbb{R}$ be differentiable functions, then:*

$$\phi \bullet \mathcal{D}^{(0,p)}(g, \bar{x}, v) = \mathcal{D}^{(0,p)}(\phi \circ g, \bar{x}, v)$$

4.2 Semantics

The operational semantics of $t^{(n,p)}$is also parametric over n and p. It is defined by the evaluation relation $\Downarrow^{(n,p)}$. Although the defining rules of $\Downarrow^{(n,p)}$ might seem fairly complex, the reader should notice how their structure reflects the derivation rules for addition, multiplication and composition of real valued functions.

$$\frac{}{(\mathbb{D}, \bar{x}, v) \vdash A \Downarrow^{(n,p)} A} \quad \text{(AD-REAL)}$$

The evaluation of unknowns is now extended to also evaluate derivatives accordingly:

$$\frac{}{(\mathbb{D}, \bar{x}, v) \vdash \mathtt{u}_{m,k} \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(\mathtt{d}^{(k)} x_m, \bar{x}, v)} \quad \text{(AD-UNK)}$$

Addition in $t^{(n,p)}$is simply defined as matrix addition:

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow^{(n,p)} A \quad (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \Downarrow^{(n,p)} B}{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \oplus \tau_2 \Downarrow^{(n,p)} A + B} \quad \text{(AD-ADD)}$$

Multiplication in $t^{(n,p)}$is defined recursively over the parameter n.

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow^{(0,p)} \bar{a} \quad (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \Downarrow^{(0,p)} \bar{b}}{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \otimes \tau_2 \Downarrow^{(0,p)} \bar{a} \star \bar{b}} \quad \text{(AD-MULT-0)}$$

$$\frac{\begin{array}{c}(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow^{(n+1,p)} A \\ (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \Downarrow^{(n+1,p)} B \\ (\mathbb{D}, \bar{x}, v) \vdash \Delta(A) \otimes I(B) \Downarrow^{(n,p)} C \\ (\mathbb{D}, \bar{x}, v) \vdash \Delta(B) \otimes I(A) \Downarrow^{(n,p)} D\end{array}}{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \otimes \tau_2 \Downarrow^{(n+1,p)} \left| \begin{array}{c} A_0 \star B_0 \\ C + D \end{array} \right|} \quad \text{(AD-MULT-N)}$$

Composition (the application of primitive functions) is also defined recursively:

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow^{(0,p)} \bar{a} \qquad a_0 \in \mathbf{dom}(\phi)}{(\mathbb{D}, \bar{x}, v) \vdash \phi\ \tau \Downarrow^{(0,p)} \phi \bullet \bar{a}} \quad \text{(AD-COMP-0)}$$

The general rule also relies on the definition of multiplication:

$$\frac{\begin{array}{c}(\mathbb{D}, \bar{x}, v) \vdash \tau \Downarrow^{(n+1,p)} A \qquad A_{0,0} \in \mathbf{dom}(\phi) \\ (\mathbb{D}, \bar{x}, v) \vdash (\phi'\ I(A)) \otimes \Delta(A) \Downarrow^{(n,p)} B\end{array}}{(\mathbb{D}, \bar{x}, v) \vdash \phi\ \tau \Downarrow^{(n+1,p)} \left| \begin{array}{c} \phi \bullet A_0 \\ B \end{array} \right|} \quad \text{(AD-COMP-N)}$$

It remains to show that these rules are actually meaningful and compute the derivatives of a function *correctly*.

5. Correctness of $t^{(n,p)}$

Recall, that theorem 1 is defined about a t-term that is lifted into a $t^{(n,p)}$-term. Thus it can be proven by structural induction over t. To do so, we show that if the property holds for every sub-term of a term it also holds for the term itself.

Case 1 ($\tau \equiv r$)**.** *By definition of lifting, we know that* $\lceil r \rceil_p^n = \left|a_{i,j}\right|$ *with* $a_{0,0} = r$ *and* $a_{i,j} = 0$ *if* $(i,j) \neq (0,0)$. *By definition of* $\rightsquigarrow$*, it holds that* f *must be the constant function* $f(\bar{x}, v) = r$ *and thus* $\partial_j \mathtt{d}^{(i)} f(\bar{x}, v) = 0$ *if* $(i,j) \neq (0,0)$.

Therefore: $a_{i,j} = \mathcal{D}^{(n,p)}(f, \bar{x}, v)_{i,j}$ *for any* $\bar{x}, v$ □

Case 2 ($\tau \equiv \mathtt{u}_{m,k}$)**.** *We know by rule* AD-UNK *that:*

$$(\mathbb{D}, \bar{x}, v) \vdash \lceil \mathtt{u}_{m,k} \rceil_p^n \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(\mathtt{d}^{(k)} x_m, \bar{x}, v)$$

By definition of $\rightsquigarrow$ *it follows that* $f(\bar{x}, v) = \mathtt{d}^{(k)} x_m(v)$.

Therefore: $\mathcal{D}^{(n,p)}(\mathtt{d}^{(k)} x_m, \bar{x}, v) = \mathcal{D}^{(n,p)}(f, \bar{x}, v)$ □

Case 3 ($\tau \equiv \tau_1 \oplus \tau_2$)**.** *We know by Lemma 1 and* $(\mathbb{D}, \bar{x}, v) \vdash \tau \rightsquigarrow f$ *that both* τ_1 *and* τ_2 *are well-formed (and compute a function each):*

$$(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \rightsquigarrow g \wedge (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \rightsquigarrow h$$

Application of the induction hypothesis to τ_1 *and* τ_2 *and insertion into* AD-ADD *yields:*

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^n \Downarrow^{(n,p)} G \quad (\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_2 \rceil_p^n \Downarrow^{(n,p)} H}{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \oplus \tau_2 \rceil_p^n \Downarrow^{(n,p)} G + H}$$

$$\text{Where} \qquad G = \mathcal{D}^{(n,p)}(g, \bar{x}, v) \qquad H = \mathcal{D}^{(n,p)}(h, \bar{x}, v)$$

Since $G + H = \mathcal{D}^{(n,p)}(g + h, \bar{x}, v)$ *and (by definition of* t*)* $f = g + h$*:*

$G + H = \mathcal{D}^{(n,p)}(g + h, \bar{x}, v) = \mathcal{D}^{(n,p)}(f, \bar{x}, v)$ □

To prove the correctness of multiplication, we need to take two steps. First, we will show the correctness in case of $n = 0$. Afterwards we apply natural induction to show the general case.

Case 4 ($\tau \equiv \tau_1 \otimes \tau_2 \quad n = 0$)**.** *As for the addition, we notice that by Lemma 1* $(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \otimes \tau_2 \rightsquigarrow f$ *implies that:*

$$(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \rightsquigarrow g \wedge (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \rightsquigarrow h \wedge f = g \times h$$

Therefore, because both terms are well-formed, we can insert the induction hypothesis into rule AD-MULT-0 *and get:*

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(g, \bar{x}, v) \quad (\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_2 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(h, \bar{x}, v)}{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \otimes \tau_2 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(g) \star \mathcal{D}^{(0,p)}(h)}$$

And by Lemma 2 and $f = g \times h$*:*

$(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \otimes \tau_2 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(f, \bar{x}, v)$ □

Case 5 ($\tau \equiv \tau_1 \otimes \tau_2$, general case)**.** *In the general case, we make the same observation about* f, g *and* h*:*

$$(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \rightsquigarrow g \wedge (\mathbb{D}, \bar{x}, v) \vdash \tau_2 \rightsquigarrow h \wedge f = g \times h$$

Additionally, we know by the application of the structural induction hypothesis:

$$(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^{n+1} \Downarrow^{(n+1,p)} \mathcal{D}^{(n+1,p)}(g, \bar{x}, v)$$
$$(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_2 \rceil_p^{n+1} \Downarrow^{(n+1,p)} \mathcal{D}^{(n+1,p)}(h, \bar{x}, v)$$

As we have seen earlier, Δ *and* I *map matrices from* $t^{(n+1,p)}$ *into* $t^{(n,p)}$*:*

$$\Delta(\mathcal{D}^{(n+1,p)}(g, \bar{x}, v)) = \mathcal{D}^{(n,p)}(g', \bar{x}, v)$$
$$I(\mathcal{D}^{(n+1,p)}(g, \bar{x}, v)) = \mathcal{D}^{(n,p)}(g, \bar{x}, v)$$

Application of the structural induction to τ_1 *and* τ_2 *yields:*

$$(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^{n+1} \Downarrow^{(n+1,p)} \mathcal{D}^{(n+1,p)}(g)$$
$$(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_2 \rceil_p^{n+1} \Downarrow^{(n+1,p)} \mathcal{D}^{(n+1,p)}(h)$$

When we set $G = \mathcal{D}^{(n+1,p)}(g)$ *and* $H = \mathcal{D}^{(n+1,p)}(h)$*, the natural induction hypothesis yields:*

$$(\mathbb{D}, \bar{x}, v) \vdash \Delta(G) \otimes I(H) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(g' \times h)$$
$$(\mathbb{D}, \bar{x}, v) \vdash \Delta(H) \otimes I(G) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(h' \times g)$$

If we apply these results to rule AD-MULT-N *we can see that:*

$$\frac{\begin{array}{c}(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^{n+1} \Downarrow^{(n+1,p)} G \\ (\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_2 \rceil_p^{n+1} \Downarrow^{(n+1,p)} H \\ (\mathbb{D}, \bar{x}, v) \vdash \Delta(G) \otimes I(H) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(g' \times h) \\ (\mathbb{D}, \bar{x}, v) \vdash \Delta(H) \otimes I(G) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(h' \times g)\end{array}}{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \otimes \tau_2 \rceil_p^n \Downarrow^{(n+1,p)} \left|\begin{array}{c} H_0 \star G_0 \\ \mathcal{D}^{(n,p)}(h \times g' + g \times h')\end{array}\right|}$$

Thus we can apply calculus to the derivation of products: $\mathcal{D}^{(n,p)}(h \times g' + g \times h') = \mathcal{D}^{(n,p)}((h \times g)') = \mathcal{D}^{(n,p)}(f')$ □

A similar technique can be used to proof correctness over composition. Again, we start with the basic case $n = 0$:

Case 6 ($\tau \equiv \phi\ \tau_1 \quad n = 0$)**.** *Again, Lemma 1 provides us with the guarantee that* τ *and thus* τ_1 *is well-formed:*

$$(\mathbb{D}, \bar{x}, v) \vdash \phi\ \tau_1 \rightsquigarrow f \Rightarrow (\mathbb{D}, \bar{x}, v) \vdash \tau_1 \rightsquigarrow g \wedge f = \phi \circ g$$

Therefore, we can apply the structural hypothesis and Lemma 3 to rule AD-COMP-0*:*

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \lceil \tau_1 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(g, \bar{x}, v)}{(\mathbb{D}, \bar{x}, v) \vdash \lceil \phi\ \tau_1 \rceil_p^0 \Downarrow^{(0,p)} \mathcal{D}^{(0,p)}(\phi \circ g, \bar{x}, v)}$$

and since: $\mathcal{D}^{(0,p)}(f) = \mathcal{D}^{(0,p)}(\phi \circ g)$ □

Case 7 ($\tau \equiv \phi\ \tau_1$, general case)**.** *As usual we start with our well-formedness observation:*

$$(\mathbb{D}, \bar{x}, v) \vdash \phi\ \tau_1 \rightsquigarrow f \Rightarrow (\mathbb{D}, \bar{x}, v) \vdash \tau_1 \rightsquigarrow g \wedge f = \phi \circ g$$

This allows us to apply the structural induction hypothesis:

$$\tau_1 \Downarrow^{(n+1,p)} G$$
$$G = \mathcal{D}^{(n+1,p)}(g)$$

Also, by definition of Δ and I:

$$\Delta(G) = \mathcal{D}^{(n,p)}(g')$$
$$I(G) = \mathcal{D}^{(n,p)}(g)$$

If we apply our observations about Δ and I from case 5 to our natural induction hypothesis, we see:

$$(\mathbb{D}, \bar{x}, v) \vdash \phi'(I(G)) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}(\phi' \circ g)$$

As we have already shown, multiplication is also correct. Thus (by application of the chain-rule, since ϕ is a single-argument function):

$$(\mathbb{D}, \bar{x}, v) \vdash \mathcal{D}^{(n,p)}(\phi' \circ g) \otimes \Delta(G) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}((\phi \circ g)')$$

Applying both results to AD-COMP-N*, we can conclude that:*

$$\frac{(\mathbb{D}, \bar{x}, v) \vdash \tau_1 \Downarrow^{(n+1,p)} G \qquad (\mathbb{D}, \bar{x}, v) \vdash \phi'\ I(G) \otimes \Delta(G) \Downarrow^{(n,p)} \mathcal{D}^{(n,p)}((\phi \circ g)')}{(\mathbb{D}, \bar{x}, v) \vdash \phi\ \tau \Downarrow^{(n+1,p)} \begin{vmatrix} \phi \bullet G_0 \\ \mathcal{D}^{(n,p)}((\phi \circ g)') \end{vmatrix}}$$

□

With this proof completed, we can state that a $t^{(n,p)}$term can be evaluated to yield any wanted total derivative and all partial derivatives of a function. Naturally, the programmer remains responsible to ensure the correctness' premise, the correspondence between the implied (differentiable) function f and the given term t for the used interval $\mathbb{D}$.

6. Implementation

In this section we will assume that equations have to be solved in a concrete host language (like Java, C, Haskell etc.) and t-terms are compiled into this host-language before lifting.

6.1 jdae

$t^{(n,p)}$has been implemented in a general purpose DAE library called jdae[2]. jdae offers the means to implement models directly in form of Java-classes. Those classes can be composed in any possible way to define a system of equations at runtime. jdae then also offers facility to index-reduce and simulate the global model.

The choice fell to Java mostly because of its bytecode access that allows for a simple implementation of the runtime specialization. We assume that $t^{(n,p)}$can be implemented quite naturally in any other language like ANSI-C or C++ as well.

[2] `https://github.com/choeger/jdae`

6.2 Basic Operations

Lifting into $t^{(n,p)}$can be implemented directly rather easily: Most languages provide the means to dynamically create and modify a matrix of real-values at runtime (either as an array or a list of floating point numbers). Lifting a t-term can thus be implemented by introducing n and p as object-level parameters into the built-in operations addition, multiplication, composition, unknown loading and constant creation. So before one evaluates a $t^{(n,p)}$term for a given n and p in the host language, one simply passes these arguments towards the term.

Neither addition nor constants render any problems: Filling a matrix with zeros but for the top-most, left-most element should be as simple to implement as matrix addition.

Loading an unknown is a slightly more complex case. Here, the implementation carefully needs to provide the different derivatives, depending on the context. First, one needs to take into account that higher order derivatives might be present implicitly in a term: If for example, one has compiled the equation $a(v)^2 + b(v)^2 = 1$ and wants to evaluate it as a $t^{(1,1)}$ term, then the implicit presence of a' yields $\bar{x} = (a, a')$. So in that case, the result of loading a by $\mathtt{u}_{1,0}$ should yield:

$$\begin{vmatrix} a(v) & 1 & 0 \\ a'(v) & 0 & 1 \end{vmatrix}$$

Since $t^{(n,p)}$is a language for DAE-simulation, the values of a and a' are either already known or part of an iterative solution process. It is only important to correctly reflect the dependencies between different unknowns.

Implementing multiplication and composition is a different story, though. The definition of their semantics hints towards a recursive implementation strategy quite directly: If one already has to implement multiplication and composition as functions of n, then it is obviously not a problem to implement them recursively. So multiplication of a $t^{(n,p)}$term relies on the implementation of the multiplication of $t^{(n-1,p)}$ and so on.

Unfortunately, this approach is hardly efficient. Not only may it involve a large amount of data copying between the recursive calls, but it also avoids an elementary optimization:

If we take a look at the multiplication, we see that every direct application of AD-MULT-N, requires *two* recursive invocations. This would yield $\mathcal{O}(2^n)$ applications. On the other hand, we know by the General Leibniz rule that we should be able to compute the n-th total derivative of a composed function by means of a sum of n elements:

$$(f \cdot g)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} f^{(k)} g^{(n-k)}$$

This can be achieved for $t^{(n,p)}$-terms as well: It is easy to observe that the *pattern* of primitive calculations $(+, \times)$ does not change depending on the numbers being multiplied:

Every field of the resulting matrix of a multiplication is calculated by a sum of products of the form:

$$(A \otimes B)_{i,j} = \sum_{(q,r,s,t) \in F_{i,j}} a_{q,r} \times b_{s,t}$$
$$i \in 0 \dots n, \in 0 \dots p$$

F can be computed recursively:

$$F_{0,0} = \{(0,0,0,0)\}$$
$$F_{0,j} = \{(0,j,0,0),(0,0,0,j)\}$$
$$F_{i+1,j} = A \cup B$$
where
$$A = \{(q+1,r,s,t) | (q,r,s,t) \in F_{i,j}\}$$
$$B = \{(q,r,s+1,t) | (q,r,s,t) \in F_{i,j}\}$$

We omit a proof of correctness for this iterative implementation for brevity. It should suffice to state that F_0 is an implementation of $\star$ and the union of A and B reflects the addition in the conclusion of AD-MULT-N. The increase of the first and third indices are the iterative version of Δ.

If one *precomputes* F once for every combination of n and p, the implementation of $t^{(n,p)}$-multiplication can be handled inside a tight loop, increasing performance and decreasing memory usage. Note, that this precomputation does not need to occur prior to compilation, but can as well be done on link- or runtime. Additionally, algebraic optimization (e.g. merging of summands with the same factors) can be applied to the precomputed elements of F, finally yielding a result similar to Leibniz' formula.

A similar approach (leading to a variant of Faà di Bruno's formula) can be implemented for the composition case.

6.3 Runtime Specialization

A way to optimize the AD-operations even further is to think of the precomputed patterns F as a form of a tiny language that is interpreted at runtime to calculate the element of the result matrix. This view yields an interesting result: As every partial derivative in the result matrix is computed in the same way (only differing in the column), there are essentially only two different methods for every operation and every concrete n (namely to compute the n-th total derivative and any given partial derivative of it).

For any concrete n, the operations remain constant for any evaluation. So *after* index reduction we can create specialized methods for every required n (in fact, we can even create some beforehand, e.g. $n = 0$). This kind of *runtime specialization* allows us to eliminate the interpretative overhead generated by the application of the precomputed operations. So instead of calculating a sum e.g. by means of a for-loop, we can directly issue a sum-expression that contains all summands, etc.

7. Conclusion

We have shown that differential algebraic equations can be given a compositional operational semantics. This semantics can be used to compile equations (and thus models) separately and still simulate them efficiently.

7.1 Example

The Cartesian pendulum model from section 2 can be compiled to t as follows (using residuals and setting $x \equiv \mathtt{u}_{1,0}, y \equiv \mathtt{u}_{2,0}, F \equiv \mathtt{u}_{3,0}$) :

$$\mathtt{u}_{1,0} \otimes \mathtt{u}_{1,0} \oplus \mathtt{u}_{2,0} \otimes \mathtt{u}_{2,0} \oplus -1 \quad (6)$$
$$\mathtt{u}_{1,2} \oplus \mathtt{u}_{1,0} \otimes \mathtt{u}_{3,0} \otimes -1 \quad (7)$$
$$\mathtt{u}_{2,2} \oplus (\mathtt{u}_{2,0} \otimes \mathtt{u}_{3,0} \oplus -9.81) \otimes -1 \quad (8)$$

As we have seen, index-reduction of the model requires us to differentiate equation 6 two times. To do so, we simply replace it by the following $t^{2,p}$ equation:

$$\lceil \mathtt{u}_{1,0} \otimes \mathtt{u}_{1,0} \oplus \mathtt{u}_{2,0} \otimes \mathtt{u}_{2,0} \oplus -1 \rceil_3^2 \quad (9)$$

Using this method, all equations of the model can be compiled separately and instantiated according to index reduction, without using symbolic or numeric differentiation.

7.2 Consequences

$t^{(n,p)}$allows for modular semantics of compiled models: There is no need to fallback to symbolic differentiation at any time for the computation of derivatives. This allows to instantiate compiled equations in any context and greatly reduces the size of the generated code (avoiding the scalability problem of traditional implementations).

It also enhances the expressiveness of DAE modeling by including models with fully variable structure. At the same time it allows for a clean separation of modules during development of models, increasing safety and reliability.

7.3 Related Work

The principle of n-th order automatic derivation in this work is based on the multivariate automatic differentiation by Kalman [6], which in turn is a generalization of Rall's numbers [12]. Kalman's operators V, D and L inspired Δ and I in this work. An early prototype of this work was developed using an implementation (by the apache commons project [3]) of Kalman's Derivative Structures. Also the idea of precomputed basic operations for multiplication and composition is based on that implementation.

Yet the fact that Kalman computes *all* mixed partial derivatives induces a large inefficiency for our application case: Inevitably, the containing Derivative Structure would also compute $\partial_1^2 \partial_i, \partial_1 \partial_2 \partial_i$ and so on (which are all unneeded for the semi-explicit solution of a DAE). So from the perspective of automatic differentiation, this work is a specialized implementation of Kalman's technique to a certain problem-domain.

The opposite approach of the implementation of n-th order automatic differentiation is the infinite computation of total derivatives as Karczmarczuk demonstrated it in [7]. The elegant, recursive style in this work inspired the formulation of the operational semantics of $t^{(n,p)}$. Albeit, Karczmarczuk does neither handle partial derivatives nor provide an iterative implementation.

Modularity of modeling languages has been researched e.g. by Zimmer [13] and Furic [3]. The former preferred an

[3] `http://commons.apache.org/math`

after-compilation approach to "rescue" as much modularity as possible, while the latter restricts the modeling formalism itself in a way that renders most symbolic manipulation needless. Both approaches differ from this work, since $t^{(n,p)}$ allows to retain *complete* modularity without sacrificing certain modeling techniques.

Another strategy is to embed a modeling language into an established general purpose language as shown by Giorgidze and Nilsson in [4]. This can even go so far as to formally define a complete host language specialized for this task, as Broman's Modelyze [2]. Both lead to modular, formally defined modeling semantics (either by inheriting from an established language or by formally defining the host language).

The main difference to our work lies in the treatment of equations: There, they are handled as data structures in the host language and interpreted (or JIT-compiled) for simulation, while in our case they can be directly translated into terms of the target language.

Finally, it should be noted that variable structure systems have been researched for a while now. In [9], Mehlhase presents an approach for systems with finite (in fact small) amounts of modes, where the symbolic manipulation can be applied to each mode separately (yielding efficient simulation code for every mode). In [14], Zimmer avoids compilation completely and shows, how runtime index reduction can concisely express certain simulation scenarios.

8. Future Work

As simulation performance is paramount, it is an obvious research topic for $t^{(n,p)}$. Simulating a set of real-world models and comparing the performance to existing implementations should be an interesting field of study.

It is an open question, how efficient index-reduction can be applied in the case of structurally variable models. As this is a non-trivial problem, any efficient solution will probably make use of an incremental approach. But until now it remains unclear, how such an approach might be implemented.

t is obviously (due to the lack of recursion) not Turing-complete. Neither is any member of $t^{(n,p)}$. To overcome this limitation, one could extend $t^{(n,p)}$ to a Turing-complete language $e^{(n,p)}$, containing all elements of $t^{(n,p)}$ plus the simple lambda calculus, general recursion (e.g. via a fixed-point operator), non-strict conditionals etc.

For any such extension of $t^{(n,p)}$, evaluation would be defined as usual (i.e. in non-AD languages). Intuitively, the automatic differentiation still "works" as in the case of $t^{(n,p)}$. Yet, it remains an interesting question how one would formulate a corresponding proof.

A second possible extension is to think in terms of *modeling*: Here it would be interesting to see, how a model *computes* different $t^{(n,p)}$ terms. For instance one could easily define the derivation operator used in modeling languages by operating on $t^{(n,p)}$-unknowns.

Another interesting aspect of $t^{(n,p)}$ is that it opens the door for a deeper use of precompiled models in the style of FMI. Using $t^{(n,p)}$ it should even be possible to introduce the concept of external *equations* into a language like Modelica (i.e. equations that are completely hidden in a precompiled library without any limitations to their usage).

References

[1] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

[2] David Broman and Jeremy G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, Jun 2012.

[3] Sébastien Furic. Enforcing model composability in modelica. In *Proceedings of the 7th International Modelica Conference, Como, Italy*, pages 868–879, 2009.

[4] George Giorgidze and Henrik Nilsson. Mixed-level embedding and jit compilation for an iteratively staged dsl. In *Proceedings of the 19th international conference on Functional and constraint logic programming*, WFLP'10, pages 48–65, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Christoph Höger. Separate compilation of causalized equations -work in progress. In François E. Cellier, David Broman, Peter Fritzson, and Edward A. Lee, editors, *EOOLT*, volume 56 of *Linköping Electronic Conference Proceedings*, pages 113–120. Linköping University Electronic Press, 2011.

[6] Dan Kalman. Doubly recursive multivariate automatic differentiation. *Mathematics magazine*, 75(3):187–202, 2002.

[7] Jerzy Karczmarczuk. Functional differentiation of computer programs. In *ACM SIGPLAN Notices*, volume 34, pages 195–203. ACM, 1998.

[8] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[9] A. Mehlhase. A Python Package for Simulating Variable-Structure Models with Dymola. In Inge Troch, editor, *Proceedings of MATHMOD 2012*, Vienna, Austria, feb 2012. IFAC. submitted.

[10] Constantinos C Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[11] John D Pryce. A simple structural analysis method for daes. *BIT Numerical Mathematics*, 41(2):364–394, 2001.

[12] L.B. Rall. *The Arithmetic of Differentiation*. MRC TSR. Defense Technical Information Center, 1984.

[13] Dirk Zimmer. Module-preserving compilation of modelica models. In *Proceedings of the 7th International Modelica Conference, Como, Italy, 20-22 September 2009*, Linköping Electronic Conference Proceedings, pages 880–889. Linköping University Electronic Press, Linköpings universitet, 2009.

[14] Dirk Zimmer. *Equation-based Modeling of Variable-structure Systems*. PhD thesis, ETH Zürich, 2010.