An XML representation of DAE systems obtained from continuous-time Modelica models

Parrotto, Roberto; Åkesson, Johan; Casella, Fracesco

2010

An XML representation of DAE systems obtained from continuous-time Modelica models

Roberto Parrotto[1] Johan Åkesson[2,4] Francesco Casella[3]

[1]Master's student - Politecnico di Milano, Italy
roberto.parrotto@gmail.com
[2]Department of Automatic Control, Lund University and Modelon AB, Sweden
johan.akesson@control.lth.se
[3]Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
casella@elet.polimi.it
[4]Modelon AB, Lund, Sweden

Abstract

This contribution outlines an XML format for representation of differential-algebraic equations (DAE) models obtained from continuous time Modelica models and possibly also from other equation-based modeling languages. The purpose is to offer a standardized model exchange format which is based on the DAE formalism and which is neutral with respect to model usage. Many usages of models go beyond what can be obtained from an execution interface offering evaluation of the model equations for simulation purposes. Several such usages arise in the area of control engineering, where dynamic optimization, Linear Fractional Transformations (LFTs), derivation of robotic controllers, model order reduction, and real time code generation are some examples. The choice of XML is motivated by its de facto standard status and the availability of free and efficient tools. Also, the XSLT language enables a convenient specification of the transformation of the XML model representation into other formats.

Keywords DAE representation, XML design

1. Introduction

Equation-based, object-oriented modeling languages have become increasingly popular in the last 15 years as a design tool in many areas of systems engineering. These languages allow to describe physical systems described by differential algebraic equations (DAE) in a convenient way, promoting re-use of modeling knowledge and a truly modular approach. The corresponding DAEs can be used for different purposes: simulation, analysis, model reduction, optimization, model transformation, control system synthesis, real-time applications, and so forth. Each one of these activities involves a specific handling of the corresponding differential algebraic equations, by both numerical and symbolic algorithms. Moreover, specialized software tools which implement these algorithms may already exist, and only require the equations of the model to be input in a suitable way.

The goal of this paper is to define an XML-based representation of DAE systems obtained from object-oriented models written in Modelica [16], which can then be easily transformed into the input of such tools, e.g. by means of XSLT transformations.

The first requirement of this system representation is to be as close as possible to a set of scalar mathematical equations. Hierarchical aggregation, inheritance, replaceable models, and all kinds of complex data structures are a convenient means for end-users to build and manage models of complex, heterogeneous physical systems, but they are inessential for the mathematical description of its behavior. They will therefore be eliminated by the Modelica compiler in the flattening process before the generation of the sought-after XML representation. However, the semantics of many Modelica models is in part defined by user-defined functions described by algorithms working on complex data structures. It is therefore necessary to describe Modelica functions conveniently in this context.

The second requirement of the representation is to be as general as possible with respect to the possible usage of the equations, which should not be limited to simulation. A few representative examples include:

- off-line batch simulation;
- on-line real-time simulation;
- dynamic optimization [3];
- transformation of dynamic model with nonlinearities and/or uncertain parameters into Linear Fractional Representation formalism [7];
- linearization of models and computation of transfer functions for control design purposes;

3rd International Workshop on Equation-Based Object-Oriented Languages and Tools. October, 2010, Oslo, Norway.

- model order reduction, i.e., obtaining models with a smaller number of equations and variables, which approximate the input-output behavior around a set of reference trajectories [8];
- automatic derivation of direct/inverse kinematics and advanced computed torque and inverse dynamics controllers in robotic systems [6].

From this point of view, the proposed XML representation could also be viewed as a standardized interface between multiple Modelica front-end compilers and multiple symbolic/numerical back-ends, each specialized for a specific purpose.

In addition, the XML representation could also be very useful for treating other information concerning the model, for example using an XML schema (DTD or XSD) for representing the simulation results, or the parameter settings. In those cases, using a well accepted standard will result in great benefits in terms of interoperability for a very wide spectrum of applications.

Previous efforts have been registered to define standard XML-based representations of Modelica models. One idea, explored in [14, 11], is to encode the original Modelica model using an XML-based representation of the abstract syntax tree, and then process the model through, e.g., XSLT transformations. Another idea is to use an XML database for scalable and database-friendly parameterization of libraries of Modelica models [17, 15].

The goal of this paper is instead to use XML to represent the system equations at the lowest possible level for further processing, leaving the task of handling aggregated models, reusable libraries etc. to the object-oriented tool that will eventually generate the XML representation of the system. In particular, this paper extends and complements ideas and concepts first presented in [5]. A similar approach has been followed earlier by [4], but has apparently remained limited to the area of chemical engineering applications.

The paper is structured as follows: in Section 2, a definition of the XML schema describing a DAE system is given. Section 3 briefly describes a test case in which a model is exported from JModelica.org platform and imported in the tool ACADO in order to solve an optimization problem. Section 4 ends the paper with concluding remarks and future perspectives.

2. XML schema representation of DAE systems

The goal of the present work is to define a representation of a DAE system which can be easily transformed into the input format of different purpose tools and then reused. A representation as close as possible to the mathematical formulation of equations is a solution general enough to be imported from the most of the tools and neutral with respect of the possible usage. For this reason concepts such as aggregation and inheritance proper of equation based object-oriented models have to be avoided in the representation.

A DAE system consists of a system of differential algebraic equations and it can be expressed in vector form as:

$$F(\dot{x}, x, u, w, t, p) = 0 \tag{1}$$

where $\dot{x}$ are the derivatives of the states, x are the states, u are the inputs, w are the algebraic variables, t is the time and p is the set of the parameters.

The schema does not enforce the represented DAEs to have index-1, but this would be the preferable case, so that the x variables can have the proper meaning of states, i.e., it is possible to arbitrarily select their initial values. Preferring the representation of models having index 1 is acceptable considering that most of the applications for DAE models require an index-1 DAE as input. In addition, in case the equations of the original model have higher index, usually an index-1 DAE can be obtained by index reduction, so the representation of index-1 DAEs doesn't drastically restrict the possible applications range.

The formulation provided in equation (1) is very general and useful for viewing the problem as one could see it written on the paper, but it is not directly usable for inter-tools exchange of models. It is then necessary to provide a standardized mathematical representation of the DAE systems that relies on a standard technology: this justifies the choice of the XML standard as a base for our representation. Hence, a formulation that better suits with our goal is proposed.

Given the sets of the involved variables

- $\dot{x} \in \mathbb{R}^n$: vector of time-varying state derivative variables
- $x \in \mathbb{R}^n$: vector of time-varying state variables
- $u \in \mathbb{R}^m$: vector of time-varying input variables
- $w \in \mathbb{R}^r$: vector of time-varying algebraic variables
- $p \in \mathbb{R}^k$: vector of bound time invariant variables (parameters and constants)
- $q \in \mathbb{R}^l$: vector of unknown time invariant variables (unknown parameters)
- $t \in \mathbb{R}$: time variable

it is possible to define the three following different subsets for the equations composing the system. The system of dynamic equations is given by

$$F(x, \dot{x}, u, w, p, q, t) = 0 \tag{2}$$

where $F \in \mathbb{R}^{n+r}$. These equations determine the values of all algebraic variables w and state variable derivatives $\dot{x}$, given the states x, the inputs u, the parameters p and q, and the time t. The parameter binding equations are given by

$$p = G(p) \tag{3}$$

where $G \in \mathbb{R}^k$. The system of parameter binding equations is assumed to be acyclic, so that it is possible to compute all the parameters by suitably re-ordering these equation into a sequence of assignments, e.g. via topological sorting. The

DAE initialization equations are given by

$$H(x, \dot{x}, u, w, p, q) = 0. \quad (4)$$

Ideally, for index-1 systems, $H \in \mathbb{R}^{n+l}$, i.e., H provides $n+q$ additional equations, yielding a well posed initialization problem with $2n + r + l$ unknowns and $2n + r + l$ equations. The initialization system is thus obtained by aggregating the dynamic equations (2) and the initialization equations (4) and determines the values of the states, state derivatives, algebraic variables and free parameters at some initial time t_0.

2.1 General design issues

The main goal is to have a schema:

- neutral with respect of the model usage;
- easy to use, read and maintain;
- easy to extend.

To achieve the first goal a representation as close as possible to the mathematical one of the DAE is required, as discussed in the previous section. To achieve the other required properties, a design based on modularity yields a result easier to read and extend. The proposed design provides one different vocabulary (namespace) for every section of the schema. In this way, if a new section will be required, for example to represent information useful for a special purpose, and a new module can be added without modify the base schema.

The Functional Mock-up Interface for Model Exchange 1.0 (FMI 1.0)[12] has been chosen as a starting point for the schema, with the main advantage of basing the work on an already accepted standard for model exchange. The FMI 1.0 specification already provides a schema containing a representation of the scalar variables involved in the system. This schema has been extended according to our goals by adding a qualified name representation for the variable identifiers, and by appending a specification of the DAE system.

The new modules composing the schema with the corresponding namespace prefixes are:

- expressions module (`exp`)
- equations module (`equ`)
- functions module (`fun`)
- algorithms module (`fun`)

All these modules, whose detailed information are given in the next paragraphs, are imported in the FMI schema, to construct the composite schema.

2.2 FMI schema and variable definitions

The FMI standard is a result of the ITEA2 project MODELISAR. The intention is that dynamic system models of different software systems can be used together for simulation. The FMI defines an interface to be implemented by an executable called FMU (Functional Mock-up Unit). The FMI functions are called by a simulator to create one or more instances of the FMU, called models, and to run these models, typically together with other models. An FMU may either be self-integrating (co-simulation) or require the simulator to perform numerical integration. Alternatively, tools may be coupled via co-simulation with network communication. The intention is that a modeling environment can generate C-code of a dynamic system model that can be utilized by other modeling and simulation environments. The model is then distributed in packages containing the C-code of the dynamic system, an XML-file containing the definition of all variables in the model, and other model information. For the present work, the FMI XML schema for description of model variables has been reused and extended.

The FMI XML schema already provides elements and attributes to represent general information about the model, such as name, author, date, generating tool, vendor annotations, but the core is the representation of the scalar variables defined in the model. It is important to notice that the FMI project is developed for the exchange of models for simulation purpose only, and not all the information present in the schema should be used in our case. Thus it is necessary to point out how to correctly use it for the purposes of this work. Firstly, the FMI schema allows the definition of Real, Integer, Boolean, String and Enumeration scalar variables. In our case, the equations (2) - (4) are all real-valued, and all time varying variables are real variables. The scalar variables definition provided by the FMI XML schema also includes attributes describing the causality (input, output, internal) and variability (constant, parameter, discrete, continuous) of the variable. Since our representation is concerned with continuously time varying DAEs only, then the definition of discrete variables should not be allowed. A full documentation of the FMI XML schema is available in [12].

The proposed representation should be neutral with respect to the application context. This also means that variable identifiers should be represented in a general way. It may happen that the tool exporting the model accepts identifiers with special characters that the importing tool does not allow. Furthermore, in the definition of user-defined functions (see a detailed discussion in Section 2.4) more complex types than scalar variables, such as array and records, are allowed. The index of an array can be a general expression, and representing the array's element by a string, e.g. `"x[3*5]"`, would require to write an ad-hoc parsing module in the importing tools. In the same manner the exporting tool can support a notation to describe array subscripts or record fields that is different from the one used by the importing tool.

For all these reasons a structured representation for qualified names, that includes only the necessary information and avoid language dependent notations is introduced. A complex type `QualifiedName` is then defined and it will be used as a standard representation for names in all the schema. The `QualifiedName` complex type expects that the identifier is broken in a list of parts, represented by `QualifiedNamePart` elements. `QualifiedNamePart` holds a string attribute "name"

and an optional element `ArraySubscripts`, to represent the indices of the array element. `ArraySubscripts` elements provide a list of elements, one for each index of the array (e.g. a matrix has an `ArraySubscripts` element with two children). Each index is generally an expression, represented by `IndexExpression`, but usually languages support definition of array variables with undefined dimensions, represented by an `UndefinedIndex` element. Conventionally, the first element of an array has index 1. In the proposed representation, definition of array variables is allowed only in user-defined functions.

Hence, the original representation of scalar variables provided by the FMI XML schema is extended in order to support the definition of variable names as qualified names, that will be the standard representation of identifiers in the whole schema.

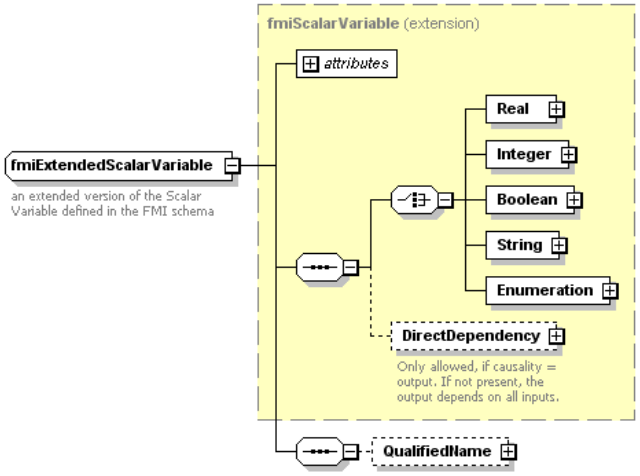

Figure 1. Scalar variable definition extended from the original FMI definition

2.3 Expressions

All the expressions are collected in the `exp` namespace. The elements in the `exp` namespace represent all the mathematical scalar expressions of the system:

- basic algebraic operators like `Add`, `Mul`, `Sub`, `Div` and the factor function `Pow`.
- Basic logical comparison operators like `>`, `>=`, `<`, `<=`, `==`.
- Basic logical operators like `And`, `Or` and `Not`.
- Built-in trigonometric functions (`Sin`, `Cos`, `Tan`, `Asin`, `Acos`, `Atan`, `Atan2`), hyperbolic functions (`Sinh`, `Cosh`, `Tanh`), the exponential function (`Exp`), the logarithmic functions (`Log`, `Log10`), the square root function (`Sqrt`), the absolute value function `Abs`, the sign function `Sign`, the `Min` and the `Max` function.
- The derivative operator `Der`.
- Function calls referring to user-defined functions.
- Variable identifiers, including the time variable.
- Real, integer, boolean, string literals.

In addition to the previous basic expressions, some special non-scalar expressions are included in the `exp` namespace: `Range`, `Array`, `UndefinedDimension` and `RecordConstructor`.

The `Range` element defines an interval of values and it can be used only in for loop definitions, inside algorithms of user-defined functions or as an argument of array constructors.

Array variable definitions and uses are allowed only within user-defined functions. It is possible to use the element `UndefinedDimension` in array variable definitions when the dimension is not known a priori. The `Array` element can be used as a constructor of an array of scalar variables in the left hand side of user-defined function call equations. Multidimensional arrays can be built by iteratively applying the one-dimensional array constructor.

As for arrays, record variables can be defined and used only in user-defined functions. The `RecordConstructor` element can be used in the left hand side of user-defined function calls, where it should be seen as a collection of scalar elements. Both record variables used in functions and record constructors used in the left hand side of equations should be compatible with a given definition of record type. The `RecordList` element, that is referenced in the main schema, should contains the definition of all the records used in the XML document, each one stored in a different `Record` element. All the elements and complex types relevant to records definition are stored in the `fun` namespace, since they are mostly related to the use of functions.

The detailed explanation of how to use `Array` and `RecordConstructor` in the left hand side of a user-defined function call equations is given in Section 2.4.

In the design of the schema, whenever a valid element is supposed to be a general expression, a wildcard element in the `exp` namespace is used, in order to simplify the representation extensibility. As a result, when a new expression is needed, it is sufficient to create a new element in the `exp` vocabulary and it will be automatically available in all the rest of the schema.

2.4 Functions

A function is a portion of code which performs a procedural computation and is relatively independent of the remaining model. A function is defined by:

- input variables, possibly with default values
- output variables
- protected variables (i.e. variables visible only within the context of the function)
- an algorithm that computes outputs from the given inputs, possibly using protected variables.

The algorithm can operate on structured variables such as arrays and records, e.g. by means of `for` loops. Differently from the variables used in equations, which can always be expressed as scalars, it is then required that input, output and protected variables of a function can also be arrays or

records, so that the algorithm can keep its original structure.

Whereas in the formulation of the equations (2) - (4) only scalar variables are involved, a detailed discussion on the use of calls for any possible cases in which the function involves non-scalar inputs or outputs is then required.

Function calls with non-scalar inputs

If an input of a function is not a scalar, it will be represented by keeping its structure, possibly using array or record constructors, but populating it with its scalar elements. In this way, it is possible to keep track of the structure of the arguments, which can then be mapped to efficient data structures in the target code.

For example, given the following definition of a record R and a function $F1$:

```
Record R
  Real X;
  Real Y[3];
End R;

Function F1
  Input R X;
  Output Real Y;
End F1;
```

A function call to $F1$ may be used as an expression in this case, since the function has only one scalar output.

```
F(R(x,{y[1], y[2], y[3]})) - 3 = 0
```

where x, y[1], y[2], y[3] are real scalar variables, `R(args)` denotes a constructor for the `R` record type, and `{var1, var2,...,varN}` represents an array constructor.

Function calls with a single non-scalar output

Auxiliary variables can be introduced to handle this case, making it possible to always have scalar equations and at the same time avoiding unnecessary duplicated function calls.

Considering the following definition of the function $F2$:

```
Function F2
  Input Real X;
  Output Real Y[3];
End F2;
```

The equation $x + F(y) * F(z) = 0$ (a scalar product) can be mapped to:

```
({aux1, aux2, aux3}) = F(y);
({aux4, aux5, aux6}) = F(z);
x + aux1*aux4 + aux2*aux5 + aux3*aux6 = 0
```

where y and z are real scalar variables.

Similarly the equation $y+F(x)-F(-3*x) = 0$, where y is an array of three real elements is mapped to:

```
({aux1, aux2, aux3}) = F(x);
({aux4, aux5, aux6}) = F(-3*x);
y[1] + aux1 - aux4 = 0;
y[2] + aux2 - aux5 = 0;
y[3] + aux3 - aux6 = 0;
```

This strategy also applies to arguments using records, or combinations of arrays and records.

Auxiliary variables are here treated as all the other scalar variables, including their declaration.

Function calls with multiple outputs

In this case, the function calls can be invoked in the following form:

$$(out1, out2, ..., outN) = f(in1, in2, ...inM) \quad (5)$$

where `out1, out2, ..., outN` can be scalar variable identifiers, array or record constructors populated with scalar variables identifiers, empty arguments, or any possible combination of these elements. So, it is not possible to write any expression on the left-hand side, nor to put the equation in residual form. Rather, this construct is used as a mechanism dedicated to handle function calls with multiple outputs while preserving the scalarized structure of system of equations.

Function with multiple outputs, cannot be used in expressions.

Given the following definition of a record type `R1` and a function `F3`:

```
Record R1
  Real X;
  Real Y[2,2];
End R1;

Function F3
  input Real x;
  output Real y;
  output R1 r;
End F3;
```

an example of call to the function $F3$ is

```
(var1,R1(var2,{{var3,var4},{var5,var6}}))
  = F1(x)
```

where `x`, `var1`, `var2`, `var3`, `var4`, `var5`, `var6`, `var7` are real scalar variables.

The proposed representation of function calls is preferable to a full scalarization of the arguments,which does not preserve any structure, and thus would require multiple implementations for the same function, e.g. if it is called in many places with different array sizes of the inputs. This solution would lead to less efficient implementations in most target languages.

Concerning the XML schema implementation, all the elements and complex types regarding user-defined function are collected in the `fun` namespace.

The main element of the `fun` namespace is `Function`, which contains the whole definition of the function, including the name, three lists of variables (respectively outputs, inputs and protected variables), the algorithm and, optionally, the definition of inverse and derivative functions. `OutputVariable`, `InputVariable` and `ProtectedVariable` elements are defined by means of the `FunctionVariable` complex type.

It is allowed, but not mandatory, to embed the definition of possible inverse and derivative functions in the `InverseFunction` and `DerivativeFunction` elements of a function definition. The information stored in

these two elements could be used for optimization purposes by the importing tool.

The elements and complex types used in the description of algorithms are defined in a different schema module than the `Function` element, but also under the `fun` namespace. The allowed statements are:

- assignments
- conditional `if` statements with `elseif` and `else` branches
- `while` and `for` loops
- function calls of user-defined functions

2.5 Equations

Complex types and elements related to the equations of the DAE system are collected under the `equ` namespace.

Once the expressions have been defined, mapping the mathematical formulation of the binding equations (3) to the XML schema is straightforward. In the `equ` namespace a complex type `BindingEquation` is defined. It provides an element `Parameter` of `QualifiedName` type that represents the left hand side of the equation, and a `BindingExp` element that represents the right hand side of the equation. An element `BindingEquations` represents the set of all the binding equations and it accepts a list, possibly empty, of `BindingEquation` elements defined as `BindingEquation` complex type.

Equations in residual form are represented by the complex type `AbstractEquation`. This type of equations provide a subtraction node to represent an equation in $exp1 - exp2 = 0$ form.

The initial equations set (4) is represented by the element `InitialEquation`, that collects a list, possibly empty, of `Equation` elements defined as `AbstractEquation` complex type.

The set of dynamic equations (2) is mapped to the `DynamicEquation` element. According to the considerations expressed in Section 2.4, equations resulting from a call to a function with multiple outputs are not suitable for representation in residual form. Thus a complex type for mapping an equation of the form (5) is given by the complex type `FunctionCallEquation`. The left hand side of the equation (5) is represented by a set of `OutputArgument` elements, defined by the `FunctionCallLeft` complex type, that can have as children scalar variable identifiers, array or record constructors populated with scalar variables identifiers, empty arguments, or any combination thereof. The right hand side is a `FunctionCall` element. It is important to notice that this element represents a set of scalar equations, one for each scalar variable in the left hand side (except for empty arguments).

Hence, the `DynamicEquations` elements contain a list of `Equation` elements of `AbstractEquation` type, which represent equations in residual form, and `FunctionCallEquation` elements, which represent equations on the form (5).

The `BindingEquation`, `DynamicEquations` and `InitialEquations` elements are directly referenced and used in the main schema.

2.6 Overall result and extensibility

Having defined the new modules, they are imported in the FMI XML schema. The elements required to be visible in the main schema are then directly referenced. The resulting overall DAE representation is given in Figure 2. In the same way, the schema could be extended by adding new information, possibly according to special purposes, developing a new separate module and referencing the main element in the XML schema, without changing the current definitions. An extension of the schema representing the formulation of optimization problems has been already developed.

Figure 2. Overall structure of the DAE XML schema

3. Test implementation

As a first implementation, an XML export module has been implemented in the JModelica.org platform [13, 1], in order to generate XML documents representing DAE systems derived from Modelica models valid with respect of the proposed schema. In addition, an extension of the XML schema for representing optimization problems has been developed and the XML code export has been implemented in the Optimica compiler, which is part of the JModelica.org compiler.

The extension for optimization problems provides elements for the representation of the objective function, the interval of time on which the optimization is performed and the constraints. The boundary values of the optimization interval, t_0 and t_f , can either be fixed or free. The constraints

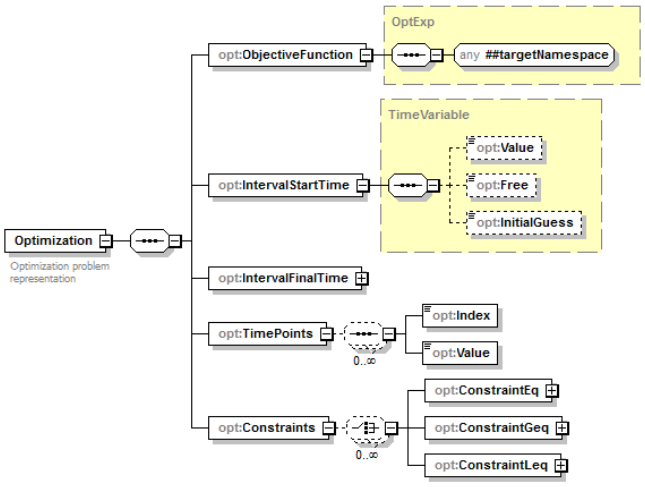

Figure 3. XML schema extension for optimization problems

include inequality and equality path constraints, but also point constraints are supported. Point constraints are typically used to express initial or terminal constraints, but can also be used to specify constraints for time points in the interior of the interval. An overall view of the extension is given in Figure 3.

Before exporting a Modelica model in XML format, a pre-processing phase is necessary. Firstly, the model should be "flattened" in order to have the system resulting from both equations of every single component and the connection equations. In this system the variables should all be scalarized. Parameters that are used to define array sizes, sometimes referred to as structural parameters, are evaluated in the scalarization and can not be changed after generation of the XML code.

Furthermore, the functions should be handled as explained in section 2.4. If the system is a higher index DAE, an index reduction operation can be performed, to obtain the final index-1 DAE.

It is interesting to notice how the XML schema has been easily implemented in the compiler. In fact, the structured representation of an XML schema is suitable to be mapped to the abstract syntax tree of the flattened model. A function that writes the corresponding XML representation has been implemented in each node class exploiting the aspect-oriented design allowed by JastAdd, used for the JModelica.org compiler construction [2]. Hence, traversing the abstract syntax tree of the flattened model is equivalent to traversing the XML document representing the same model. In the same way, importing an XML representation of the model could be done traversing the XML document and building a node of the syntax tree corresponding to each XML node. This remains to be done.

For the test case, the ACADO toolkit [10] has been chosen as importing tool. ACADO Toolkit is a software environment, not specifically related to Modelica, that collects algorithms for automatic control and dynamic optimization. It provides a general framework for using a great variety of algorithms for direct optimal control, including model predictive control, state and parameter estimation and robust optimization.

The Van der Pol oscillator model (it can be found with further explanations in [13]) has been exported from the JModelica.org platform and imported into ACADO. The goal is to solve an optimal control problem with respect to the constraint $u <= 0.75$ acting on the control signal while minimizing a quadratic cost function in the interval $t \in [0, 20]$.

The optimal control problem has been parameterized as an non-linear problem using direct multiple shooting (with condensing) and solved by an SQP method (sequential quadratic programming) based on qpOASES [9] as a QP solver.

The results are given in Figure 4. The same results can be obtained by solving the problem by means of a collocation method available in JModelica.org.

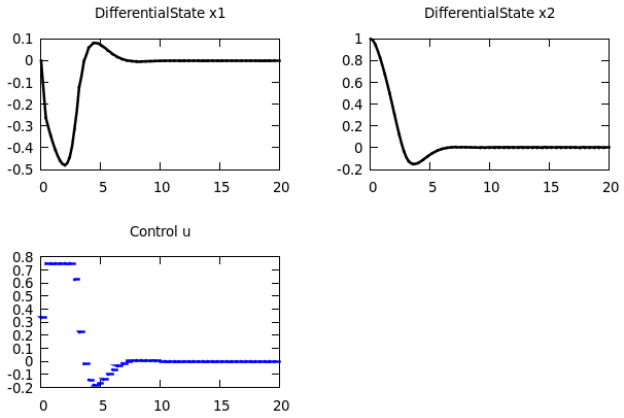

Figure 4. Van der Pol optimization problem results from ACADO

4. Conclusions and future perspectives

In this paper, an XML representation of continuous time DAEs obtained from continuous-time Modelica models has been proposed. The test implementation on the JModelica.org platform has shown the possibility to use the XML representation to export Modelica models and then reuse them in another non-Modelica tool. In the same manner, many other possible applications could be considered [6].

A future version of the schema could extend the representation to hybrid DAE systems. In this case the concept of discontinuous expressions, discrete variables, discrete equations and events should be introduced.

An interesting perspective could be to explore to which extent the proposed DAE representation could be used to describe flattened models written using other equation-based, object-oriented languages, possibly by introducing additional features that are not needed to handle models obtained from Modelica, in the same spirit of the CapeML initiative [4].

Finally, it would also be interesting to investigate the possibility to aggregate models represented by different XML documents. In this case every XML document would

represent a sub-model and an interface to allow more sub-models to be connected should be designed.

References

[1] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, January 2010. Doi:10.1016/j.compchemeng.2009.11.011.

[2] Johan Åkesson, Torbörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1):21–38, 2010.

[3] L.T. Biegler, A.M. Cervantes, and A. Wachter. Advances in simultaneous strategies for dynamic process optimization. *Chemical Engineering Science*, 57(4):575–593, 2002.

[4] Christian H. Bischof, H. Martin Bücker, Wolfgang Marquardt, Monika Petera, and Jutta Wyes. Transforming equation-based models in process engineering. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering, pages 189–198. Springer, 2005.

[5] F. Casella, F. Donida, and Åkesson. An XML representation of DAE systems obtained from Modelica models. In *7th Modelica conference*, September, 20-22 2009.

[6] F. Casella, F. Donida, and M. Lovera. Beyond simulation: Computer aided control system design using equation-based object oriented modelling for the next decade. In *2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, July, 8 2008.

[7] F. Casella, F. Donida, and M. Lovera. Automatic generation of LFTs from object-oriented non-linear models with uncertain parameters. In *6th Vienna International Conference on Mathematical Modeling*, February, 11-13 2009.

[8] Francesco Casella, Filippo Donida, and Gianni Ferretti. Model order reduction for object-oriented models: a control systems perspective. In *Proceedings MATHMOD 09 Vienna*, pages 70–80, Vienna, Austria, Feb. 11–13 2009.

[9] H.J. Ferreau, H.G. Bock, and M. Diehl. An online active set strategy to overcome the limitations of explicit MPC. *International Journal of Robust and Nonlinear Control*, 18(8):816–830, 2008.

[10] KU Leuven. ACADO toolkit Home Page. `http://www.acadotoolkit.org/`.

[11] J. Larsson. A framework for simulation-independent simulation models. *Simulation*, 82(9):563–379, 2006.

[12] Modelisar. Functional Mock-up Interface for Model Exchange, 2010. `http://www.functional-mockup-interface.org`.

[13] Modelon AB. JModelica Home Page, 2009. `http://www.jmodelica.org`.

[14] A. Pop and P. Fritzson. ModelicaXML: A Modelica XML representation with applications. In *3rd Modelica conference*, November, 3-4 2003.

[15] U. Reisenbichler, H. Kapeller, A. Haumer, C. Kral, F. Pirker, and G. Pascoli. If we only had used XML... In *5th Modelica conference*, September, 4-5 2006.

[16] The Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, 2009. `http://www.modelica.org/documents/ModelicaSpec32.pdf`.

[17] M. Tiller. Implementation of a generic data retrieval API for Modelica. In *4th Modelica conference*, March, 7-8 2005.