ISSN 0280-5316
ISRN LUTFD2/TFRT--5865--SE

An XML Representation of DAE Systems obtained from Continuous-time Modelica Models

Roberto Parrotto

Department of Automatic Control
Lund University
November 2010

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
November 2010

Document Number
ISRN LUTFD2/TFRT--5865--SE

Author(s)
Roberto Parrotto

Supervisor
Johan Åkesson Autmatic Control, Lund
Karl-Erik Årzén Automatic Control. Lund (Examiner)

Sponsoring organization

Title and subtitle
An XML Representation of DAE Systems obtained from Continuous-time Modelica Models (En XML-representation för DAEer genererade från kontinuerliga Modelica modeller)

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

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
109

Recipient's notes

Security classification

Acknowledgments

This thesis has been developed during an amazing experience in Lund, Sweden, where I have been guest at Modelon AB and at the Dept. of Automatic Control of Lund University. I will never thank enough the people who helped me to organize this adventure.

First, I would like to thank my advisor, Professor Francesco Casella, who proposed me the project, Johan Åkesson, who has been my tutor during the whole period in Sweden and Professor Gianni Ferretti for suggesting my name for the project and transmitting me the interest for the topic.

I would like to thank all the guys at Modelon for the nice time, and expecially Hubertus Tummescheit for the interesting talks, Tove Bergdahl and Jesper Mattson for the kind help with the JModelica.org platform. Thanks also to Joel Andersson of K.U. Leuven for the help with the test case and the ACADO platform.

I absolutely need to thank all the friends I have met during these 8 months, who made the experience unforgettable: Aarti, Carol, Ester, Felix, Jaha, Leslie, Magda, Mariana, Reena, Ranko,Yina and I am sorry if I don't have space enough to list all of you.

Last, but by no means least, since the thesis is just the last effort of a 5 years journey, I would like to thank all the people, you know who I am talking about, who stayed at my side during the years in Cremona and Milano and expecially during the first two difficult years of this period.

Roberto

Contents

List of Figures

List of Tables

Chapter 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 work is to define an XML-based representation of DAE systems obtained from object-oriented models written in Modelica [35], 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, heterogenous physical systems, but they are inessential for the mathematical description of its behaviour. 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 [8];
- transformation of dynamic model with nonlinearities and/or uncertain parameters into Linear Fractional Representation formalism [12];
- linearization of models and computation of transfer functions for control design purposes;
- model order reduction, i.e., obtaining models with a smaller number of equations and variables, which approximate the input-output behaviour around a set of reference trajectories[13];
- automatic derivation of direct/invers kinematics and advanced computed torque and inverse dynamics controllers in robotic systems [11].

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 [29, 23], 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 XML database for scalable and database-friendly parametrization of libraries of Modelica models [36, 31].

The goal of this thesis 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 thesis extends and complements ideas and concepts first presented in [10]. A similar approach has been followed earlier by [9], but has apparently remained limited to the area of chemical engineering applications.

The report is structured as follows: in Chapter 2 a short introduction to the DAEs theory is given. Chapter 3 introduces the Modelica language. In Chapter 4 the definition of the XML schema describing a DAE system is discussed. Chapter 5 describes how to map the defined schema to Modelica models. Chapter 6 presents 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, while Chapter 7 ends the report with concluding remarks and future perspectives.

Chapter 2

Introduction to DAE systems

2.1 Definition of DAE system

The behaviour of physical dynamical processes is usually modeled via differential equations (ODE), but in many cases constraints exist on the states of the physical system, as for example Kirchhoff's laws in electrical networks. Systems consisting of both differential and algebraic equations are called differential-algebraic equations systems (DAE) or singular systems.

The most general form of a DAEs system is given by

$$\mathbf{F}(t, \mathbf{x}, \mathbf{y}') = \mathbf{0} \tag{2.1}$$

where $\delta\mathbf{F}/\delta\mathbf{y}$' may be singular. Each component of $\mathbf{y}'$ may contain a mix of dynamical and algebraic components, leading to a hard problem for finding the numerical solution. An important special case of (2.1) is the semi-explicit DAE (or ODE with constraints) :

$$\begin{aligned} \mathbf{x}' &= \mathbf{f}(t, \mathbf{x}, \mathbf{z}) \\ \mathbf{0} &= \mathbf{g}(t, \mathbf{x}, \mathbf{z}) \end{aligned} \tag{2.2}$$

where $\delta\mathbf{g}/\delta\mathbf{z}$ is supposed to be nonsingular. In this case it is possible to

decouple the differential variables $\mathbf{x}$(t) and algebraic variables $\mathbf{z}$(t) and for each instant a solution can be found by calculating $\mathbf{z}(t, \mathbf{x})$ from $\mathbf{g}(t, \mathbf{x}, \mathbf{z}) = 0$, replacing it in $\mathbf{x}' = \mathbf{f}(t, \mathbf{x}, \mathbf{z})$ and integrating the resulting ODE system.

DAE theory is much more recent than ODE theory and there are many differences that complicate the research of a numerical solution for DAE compared to ODE's one. First of all, a generally valid theorem for existence, uniqueness and continuous dependence on the data exists for ODE's solution but not for DAE's. Furthermore it is not possible to arbitrarily set the initial conditions for a DAE system, but they must satisfy the explicit constraints and the hidden constraints resulting from the derivative of the explicit constraints. Those initial conditions that hold this property are called consistent conditions. On the numerical point of view, the class of DAEs includes all ODEs, but also problems where both differentiation and integration are intertwinned in order to find the solution and the effects of the former complicates the numerical integration process. Since in the present work we are more interested in the representation of DAE systems, the problem of numerical integration is beyond our scope, but an interesting exposition can be found on [20].

2.2 Index of a DAE system

If the problem is not singular, it is possible to apply analytical differentiations to a given DAE system, repeatedly if necessary, to transform the problem to an explicit ODE system. The number of differentiations needed to obtain the ODE system is called the index of the DAE. According to the previous definition an ODE is a DAE having index 0 and the semi-explicit DAE system (2.2) is clearly index 1. DAEs with index greater than 1 are called higher index DAEs. As already explained, initial conditions must be consistent also with the hidden constraints introduced by the differentiations. It is important to note that the local linearization depends on the solution, therefore also

the index depends not only on the form of the DAE, but also on the solution. Hence, we can give a more precise definition of index.

Definition. *For the general DAE system* (2.1)*, the index along a solution* $\boldsymbol{y}(t)$ *is the minimum number of differentiations of the system which would be required to solve for* $\boldsymbol{y'}$ *uniquely in terms of* $\boldsymbol{y}$ *and* t *(i.e. to define an ODE for* $\boldsymbol{y}$*). Thus, the index is defined in terms of the overdetermined system*

$$\begin{aligned} \mathbf{F}(t, \mathbf{y}, \mathbf{y}') &= \mathbf{0} \\ \tfrac{\mathbf{dF}}{\mathbf{dt}}(t, \mathbf{y}, \mathbf{y}', \mathbf{y}'') &= \mathbf{0} \\ &\cdot \\ &\cdot \\ &\cdot \\ \tfrac{\mathbf{d^p F}}{\mathbf{dt^p}}(t, \mathbf{y}, \mathbf{y}', \mathbf{y}'', ..., \mathbf{y^{(p+1)}}) &= \mathbf{0} \end{aligned} \tag{2.3}$$

to be the smallest integer p *so that* $\boldsymbol{y'}$ *in* (2.3) *can be solved for in terms of* $\boldsymbol{y}$ *and* t*.*

The index previously defined is also named differentiation index, but it is possible to find other different definitions in the literature, for example as a measure of sensitivity of the solutions with respect to pertubations for a given problem (perturbation index, [19]) or as the size of the largest Jordan block to an infinite eigenvalue in the associated Kronecker canonical form (algebraic index) [22, par.1.2] , which is useful in the analysis of DAE with constant coefficients.

2.3 Index reduction

Direct discretization methods are limited in their usefulness to index-1 and semi-explicit index-2 DAE systems. Even though there are some results on the convergence of some methods for case of higher order DAEs, there are practical difficulties in writing robust codes that don't involve a substantial

user intervention. In order to use general purpose solvers an order reduction process is usually performed. In fact the solution of index-1 DAE (and of other special DAE forms called of Hessenberg [7, par.9.1.1]) is not much more difficult than that of stiff ordinary differential equations and fortunately many DAEs in practical applications are either index-1 or can be seen as a composition Hessenberg systems [7, ch.10] .

In order to reduce the index it is possible to differentiate the algebraic constraints, repeating the process until the problem will be index-1. Performing the index reduction by differentiations, the introduced numerical errors should be controlled in order to preserve the constraints, because of the well know drift-off phenomenon of the operation. The process leads to an over-determined system, that can be solved by the dummy derivative method, that it is explained in the classical paper by Mattsson and Söderlind [24]. There are many other numerical methods to solve the index reduction problem and more explanations can be found in [20, cap.VII.2] and [22, ch.6].

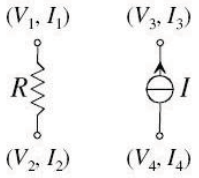
Many simulation tools use a graph-based algorithm to perform the index reduction of the DAE system [28]. This iterative procedure, known as Pantelides algorithm, establishes a minimum set of equations which must be differentiated in order to remove a structural singularity. Each time equations are differentiated, a path finding process is applied to the bi-partite graph of equations and variables to attempt to find a mapping which uniquely assigns each unknown variable in the system to an equation which can be used to calculate that variable. The algorithm will go on differentiating the system of equations until a complete assignment has been found. It is interesting to note that if the algorithm doesn't reach a solution, it is possible to prove that it is not possible to find consistent initial conditions for the given DAE system, which therefore can't be solved. This result is also known as Pantelides theorem.

2.4 Acausal approach to modeling

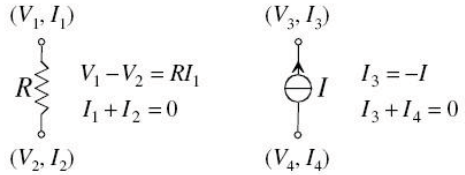
In order to design complex mathematical models it is useful to proceed by aggregation of smaller components. In the case of causal models described by ODEs it is simple to simulate aggregated models, but the topology of the resulting system usually doesn't correspond to the structure of the physical one. As a result the readability and reusability of the model will be heavily affected. Therefore, although causal models are a straightforward solution for describing control systems, an acausal approach, using DAEs, best suits the problem of designing physical models. The key idea is to describe the behaviour of every elementary physical component, or specific phenomena, by equations and to design an interface in order to connect them, always according to physical principles: when two or more components are mutually connected the effort variables are equalized and the flow variables are balanced. The complete system model, including the component equations and the connection equations, thus corresponds to a DAE system, usually called flattened model.

Example. Connection of two simple models in an electrical network.

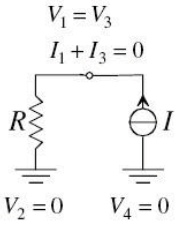
Let's consider a simple network involving the following two components :



where the effort variable is the potential V_i, while the flow variable is the current I_i.The equations describing each component are the following:



Finally, when the two components are connected to each other and to the ground, the following four equations are added to the model:



hence, the final model is described by 8 unknowns and 8 equations.

Chapter 3

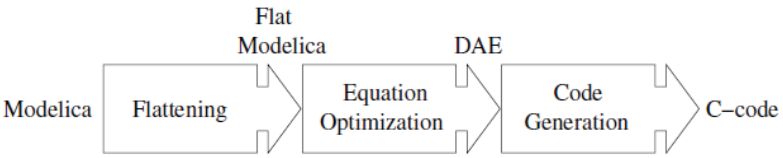
The Modelica language

3.1 Introduction to the Modelica language

Modelica is an object-oriented, declarative, multi-domain modeling language for component-oriented modeling of complex systems.

The Modelica language is widely used in industry for modeling and simulation of physical systems [34]. Example application areas include industrial robotics, automotive applications and power plants. Typical for models of such systems is that they are of large scale, commonly up to 100.000 equations. There are also extensive standard libraries for e.g., mechanical, electrical, and thermal models. The first version of Modelica was published in September 1997. The effort was targeted at creating a new general-purpose modeling language, applicable to a wide range of application domains. While several other modeling languages were available, many of them were domain-specific, which made simulation of complex multi-domain systems difficult. Based on experience obtained from designing other modeling languages, notably Dymola, [15], and Omola, [6], the fundamental concepts of object-orientation and declarative programming were adopted. The work presented in this thesis, however, is based on the latest available specification, version 3.2 [35], which was released in April 2010.

Figure 3.1: Modelica translation process



Mathematical modeling of complex physical systems requires appropriate high level languages. In particular, it is essential that a modeling language for such systems offers abstractions for structuring of models. A particularly successful paradigm has been that of object-oriented modeling. In the Modelica context, the structural concepts of object-orientation, such as classes, components and inheritance, are emphasized, rather than dynamic creation of objects and message passing.

Modelica is designed with multi-domain modeling in mind. Accordingly, the language is particularly useful for applications which involve modeling of physical phenomena from different domains. For example, in automotive applications, it is desirable to have sub-models for the combustion, the mechanical systems, the electronics and the interaction with the road. In this type of applications, Modelica serves as a unifying language in which all these sub-systems can be modeled. The primary objective of formulating a model of a complex physical system is most often simulation. That is, prediction of the response of a model, given a specified input stimuli, and the state of the model. If the model is sufficiently accurate, the simulation results may be used to draw conclusions about the behavior of the true physical system.

3.2 The Modelica translation process

The process of translating Modelica source code into a format suitable for numerical simulation/optimization algorithms can be divided into a number of steps, see Figure3.1. In the first step, the Modelica code is flattened. This

means that all component and inheritance structures are eliminated. The resulting model contains essentially a set of variables and a set of equations including both component and connection equations. The only property of the flat model that indicates its hierarchical origin is that a variable name is usually expressed as a qualified name, indicating the path of the corresponding variable. In the next step, the equations are sorted using graph-theoretical methods such as the BLT transformation, [32]. Sorting of equations are done in order to explore the structure of the model. The equations are then analyzed further and manipulated so that they can be more efficiently solved by numerical software. The output of this step is referred to as a differential algebraic equation (DAE). The DAE also represents a generic mathematical description of the original Modelica model, and may be used for different purposes. The commonest application is to generate C code, which is compiled and linked with an algorithm for numerical integration. The behavior of the system can then be simulated by executing the resulting application.

3.3 Overview of the language

Modelica is "a unified object-oriented language for physical systems modeling", [35]. As such, its most important feature are:

- Modelica supports equation-based acausal modeling, as opposed to assignment statements. Using equations, the modeler can state relations on their most natural form, without the need to solve for a particular variable. As a consequence, the data-flow direction is not determined a priori, but rather by the context of a particular component.

- Modelica can be used to express models from different domains, enabling modeling of heterogeneous systems

- Modelica is an object-oriented language. This feature enables the modeler to use powerful structuring concepts such as classes, components,

inheritance and generics.

- Modelica has strong support for component-based models, including means to connect components. This feature enables modelers to create modular models, as well as interfaces through which they can be connected.

These properties make Modelica particularly well suited for modeling of large and complex systems. For example, a well known limitation of block-based modeling is the need to solve for a particular set of variables. Even though the equations for each component in a composite model are simple and straightforward to derive, the modeler has to transform, usually by hand, the original model component equations into the standard ODE representation. For many physical systems, this transformation is often global, in the sense that all model components have to be considered simultaneously. In addition, the original structure of the model is often destroyed in the transformation. Modelica overcomes this difficulty by allowing acausal connection of model components. This approach leaves to the tool to transform the model equations into a format suitable for, for example, numerical integration. The structuring constructs of Modelica, such as classes, inheritance, generics and packages promote model reuse and development of model libraries. This, in turn, enable domain experts to encapsulate knowledge in an accessible and structured way.

The class concept is fundamental in Modelica. Apart from the built-in classes "Real", "Integer", "Boolean" and "String", classes can also be defined by the user. A Modelica class may contain local class definitions, component declarations (these two entities are referred to as elements), equations and algorithms. A component declaration corresponds to an instance of a class, which can be either a user-defined class or a built-in class, such as Real. In the latter case, the component declaration is sometimes referred to as a variable declaration. The variability of a variable can be specified. For example, a variable of type Real can be specified to be a parameter, which means that

it is constant during simulation. Elements can also be specified to be either public or protected. In the latter case, such elements are only accessible from within the class itself. In addition, Modelica supports multiple inheritance.

Equations and algorithms are used to define behavior. Physical phenomena are often modeled by mathematical equations. Equations in turn define relationships between physical variables, such as pressure, temperature, current or voltage. In addition, many physical phenomena are described by differential equations, where the variables as well as their derivatives with respect to time or space appear. Typical origins of equations are the laws of nature, e.g. the law of conservation of energy or Ohm's law. By stating equations declaratively, the need to solve for a certain variable determined by the model context, and possibly simulation environment, is eliminated. Consider e.g., Ohm's law, valid for an ideal resistor: $v = Ri$, where v is the difference in potential between the terminals, R is the resistance and i is the current. This equation can be stated in three different ways. Apart from the standard form, assuming $R \neq 0$,$i = v/R$ and $R = v/i$ also have the same mathematical meaning. This example illustrates what is a well known problem when formulating simulation models: it is often necessary to have several versions of the same model depending on how it is connected to its environment. Modelica solves this problem by enabling the user to state equations on their natural form and then leave it to the tool to transform the model into simulation code. Most realistic systems exhibit discontinuous behavior. For example, the gear box of a car have a limited number of discrete gears. Also, equipment that is controlled by switching it on and off results in models with discrete behavior. Systems containing both continuous and discrete dynamics are referred to as hybrid systems. Since many engineering problems are hybrid in this sense, Modelica offers support for expressing hybrid models. There are two constructs available in Modelica for introducing hybrid behavior. if-clauses are used to express conditional equations, i.e., based on one or many conditions, a corresponding set of equations are active

in the model. It is also possible to express instantaneous events, i.e., when a specified condition evaluates to true, some actions should be taken.

While physical phenomena are conveniently expressed by equations, there are other types of behavior that are expressed in a more natural way using algorithms. One such example is discrete-time control systems. Since it is often desirable to model not only the actual physical system, but also the associated control system, the ability to express algorithms in Modelica is important. Algorithms in Modelica can be used to express sequences of assignment statements, conditional statements and iteration.

Physical systems can often be decomposed into distinct subsystems, which are connected. By using a top-down approach, models for the subsystems can be combined to form more complex composite models. This methodology is strongly supported in Modelica. Specifically, the specialized class connector and the built-in function connect can be used to formulate structured models composed of interacting components. A connector class serves as an interface between components. If two components both have a connector of the same type, they may be connected. The interface defined by a connector class consists of a set of variables, which are either of potential type or flow type. The semantics of a connection operation is the following: When a connection is formed, the potential variables of all connected connector components are set equal, while the sum of the flow variables is set to zero. During translation of a Modelica model, equations are generated from connection statements.

For a comprehensive description of the Modelica language and its usage, see [18].

Chapter 4

An XML representation of DAE systems

4.1 Mathematical formulation of flat DAE models for an XML representation

The goal of the present work is to define a representation of a DAE system obtained from continuous-time Modelica models, 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 largest set of the tools and neutral with respect of the possible usage. For this reason concepts as aggregation and inheritance, which are proper of equation based object-oriented models, should be avoided in the representation. Furthermore, the designed representation may be used to represent acausal declarative models written in different languages, but this possibility has not been explored at the time of this thesis and it is proposed as a future perspective in Chapter 7.

A DAE system consists of a system of differential algebraic equations and

it can be expressed as:

$$F(\dot{x}, x, u, w, t, p) = 0 \tag{4.1}$$

where $\dot{x}$ is vector of derivatives of the state, x is vector of the state, u is vector of the inputs, w is vector of the algebraic variables, t is the time and p is vector 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 meaning of states and 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 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 (4.1) 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

- $x \in \mathbb{R}^n$: 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.

$$F_i(x, \dot{x}, u, w, p, q, t) = 0,\ i = 1...n + m \tag{4.2}$$

is the set of dynamic equations. Each function $F_i(x, \dot{x}, u, w, p, q, t)$ denotes a valid scalar expression in the scalar elements of its arguments, giving the scalar residual of the i-th equation. 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.

$$p_i = G_i(p),\ i = 1...k \tag{4.3}$$

is the set of parameter binding equations. Each function $G_i(p)$ denotes a valid expression in the scalar elements of its argument. 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 Tarjan's algorithm [32].

$$H_i(x, \dot{x}, u, w, p, q) = 0,\ i = 1...n + l \tag{4.4}$$

is the set of initial equations. Each function $H_i(x, \dot{x}, u, w, p, q)$ denotes a valid expression in the scalar elements of its arguments. The system formed by the dynamic equations (4.2), the parameter binding equations (4.3) and the initial equations (4.4), which has $2n + m + k + l$ equations, determines the values of the states, state derivatives, algebraic variables and parameters at some initial time t_0.

4.2 XML Schema representation

An XML document [38] is an XML file that represents an instance of a data structure. An XML Schema [39] defines the rules that an XML document should hold to be valid for a certain application. Plenty of tools and libraries are available to verify that a certain XML document is valid with respect of an XML schema. In our case the XML schema will then describe the structure of a DAE system's valid representation while the instances, i.e. the XML documents, contain the actual equations of a model exported from a certain tool. In this section a description of the XML schema is given, while a short documentation of the key concepts of XML Schema language is available in Appendix A.

4.2.1 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 paragraph. To achieve the other required properties a design based on modularity guarantees 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, a new module can be added without modifying the base schema. An example of this usage is given in section 4.4.

The Functional Mock-up Interface for Model Exchange 1.0 (FMI 1.0)[25] has been choosen 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 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 names 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:

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

All these modules, whose detailed description is given in the next paragraphs, are imported in the FMI schema, to obtain the overall result shown in Figure 4.1

4.2.2 FMI Schema and variables representation

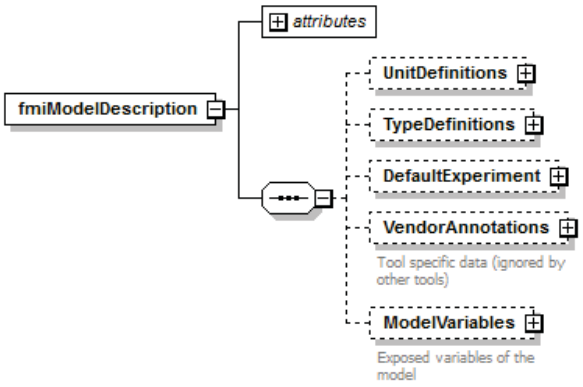
The Functional Mockup Interface definition is one result of the ITEA2 project MODELISAR [25]. The intention is that dynamic system models of different software systems can be used together for simulation. The FMI (Functional Mock-up Interface) defines an interface to be implemented by an executable called 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 shall be coupled via co-simulation with network communication. The intention is that a modelling environment can generate C-code of a dynamic system model that can be

Figure 4.1: Overall resulting schema

utilized by other modelling and simulation environments. The model is then distribuited in packages containing the C-code of the dynamic system and an xml-file containing the definition of all variables in the model and other model information.

For the sake of the present work the XML Schema is the interesting part of the FMI project to take into account. At the root-level the base FMI schema is so defined:

Figure 4.2: Overall of the original FMI schema



At the top level the FMI schema consists of:

element	*description*
attributes	Global properties of the model, such as the model name, author and generating tool
UnitDefinitions	A global list of definitions to convert display units into the units used in the model equations. These definitions are used in the xml-element "ModelVariables"
TypeDefinitions	A global list of type definitions that are utilized in "ModelVariables".
DefaultExperiment	Providing default settings for the integrator, such as stop time and relative tolerance.
VendorAnnotations	Additional data that a vendor might want to store and that other vendors might ignore.
ModelVariables	The central FMI data structure defining all variables of the model that are visible/accessible via the model functions

Table 4.2: FMI XML Schema: top level elements

A full description of every section the FMI schema is available in the official documentation [25]. The optional attribute "variableNamingConvention" can be omitted in the exported XML documents, since the FMI schema is extended to support qualified names as explained further on. The optional element "DefaultExperiment" contains information oriented to simulation applications and can also be ignored for our goal. "UnitDefinitions" and "TypeDefinitions" can be considered as optional and neutral with respect of our goal, i.e. the choice of generating these information in the XML documents will not change the meaning of the DAE representation, but it can just be useful to further document the role of the variables. "VendorAnnotations" is optional and can store information on the tool exporting the model.

More interesting for our target is the representation of the variables. "ModelVariables" element consists of a set of "ScalarVariable" elements defined as "fmiScalarVariable" complex type (Figure 4.3). A "fmiScalarVariable" represents one primitive type, like a real or integer variable. Only scalar variables are supported in the FMI schema and structured entities (like arrays or records) have to be mapped to scalars.

A description of the attributes of "fmiScalarVariable" is given in the following table:

Table 4.3: "fmiScalarVariable" attributes

attribute	*description*
name	The full, unique name of the variable.
valueReference	A handle of the variable to efficiently identify the variable value in the model interface. It is used by the C-functions of the FMU
description	An optional description string describing the meaning of the variable

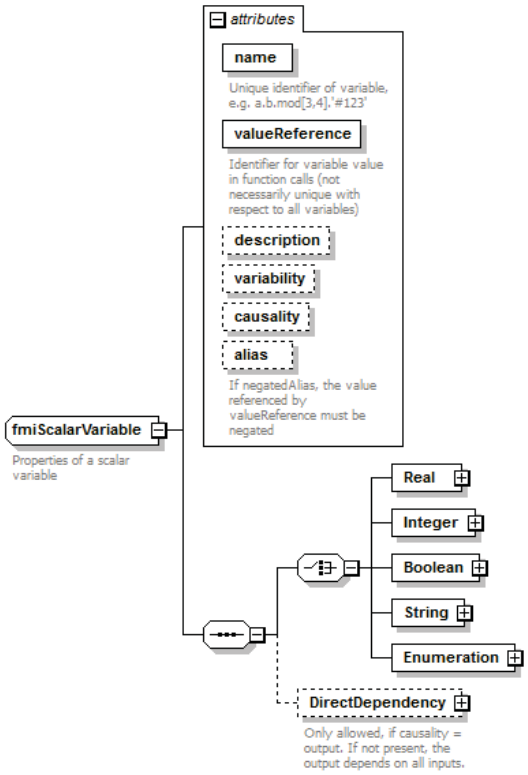
variability	Allowed variables are: • "constant": the value of the variable is fixed and does not change. • "parameter": The value of the variable does not change after initialization • "discrete": The value of the variable only changes during initialization and at event instants. • "continuous": No restrictions on value changes. Only a variable of type = "Real" can be "continuous". The default is "continuous".
causality	Defines how the variable is visible from the outside of the model. It can have one the following values: "input","output","internal","none" and the default is "internal"

alias	Enumeration that defines whether the respective variable is an alias variable. An alias variable is the result of an equation "a := b" or "a := −b" (this situation occurs very often in models built-up by connecting physical components together). In order to retrieve the value of "a" from the value of "b", the alias property is defined with this attribute and the valueIdentifier is the one from "b". Allowed enumeration values: • "noAlias": It is not an alias variable (this is the default). • "alias": The variable is an alias variable. The actual value can be set/get via the valueReference handle. • "negatedAlias": The variable is an alias variable where the variable value retrieved via the valueReference handle must be negated The alias property can be used for efficiency optimizations.

In addition to the described attributes, the "fmiScalarVariable" complex type provides elements to collect the attributes specific to the type of the variable. From Figure 4.3 it is possible to notice that the FMI schema allows "Real", "Integer", "Boolean", "String" and "Enumeration" types. An optional element "DirectDependency" can be used if the causality of the variable is set to "output".

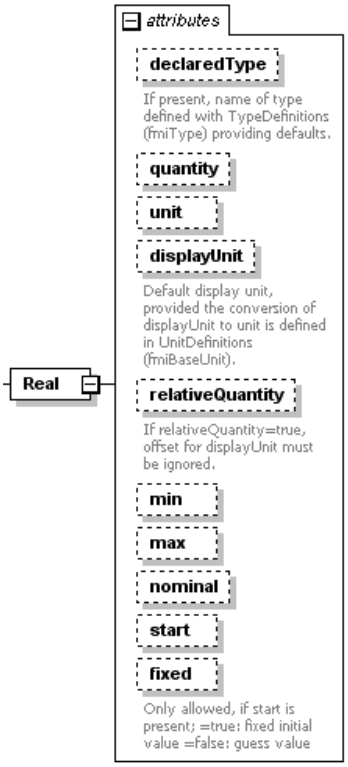
After the previous general documentation, it is important to point out

Figure 4.3: Scalar Variables representation in the original FMI schema



how to correctly use the description of scalar variables in our DAE representation. According to the formulated mathematical representation in Section 4.1, no discrete variables are allowed, so the "discrete" choice of "variability" attribute should never be used. The only allowed type for the state variables is "Real", while parameters could also be "Boolean" or "Integer". Only the attributes of "Real" element will now be described (Figure 4.4), but the representation of the types is almost identical.

Figure 4.4: Attributes of "Real" element



The “Real” element holds the following attributes:

Table 4.5: “Real” element attributes

attribute	*description*
declaredType	If present, name of type defined with TypeDefinitions.
quantity	Physical quantity of the variable, e.g., “Angle”, or “Energy”
unit	Unit of the variable that is used for the model equations, e.g., “N”.
displayUnit	Default display unit. The conversion to the “unit” is defined with the element “fmiModelDescription / UnitDefinitions”.
relativeQuantity	If this attribute is true, then the “offset” of “displayUnit” must be ignored (e.g. 10 degree Celsius = 10 Kelvin if “relativeQuantity = true” and not 283 Kelvin).
min	Minimum value of variable (variable $\geq$ min). If not defined, the minimum is the largest negative number that can be represented on the machine.
max	Maximum value of variable (variable $\leq$ max). If not defined, the maximum is the largest positive number that can be represented on the machine.
nominal	Nominal value of variable. If not defined and no other information about the nominal value is available, then nominal = 1 is assumed.
start	Initial value of variable.All constants and independent parameters must have a start value in the xml-file.

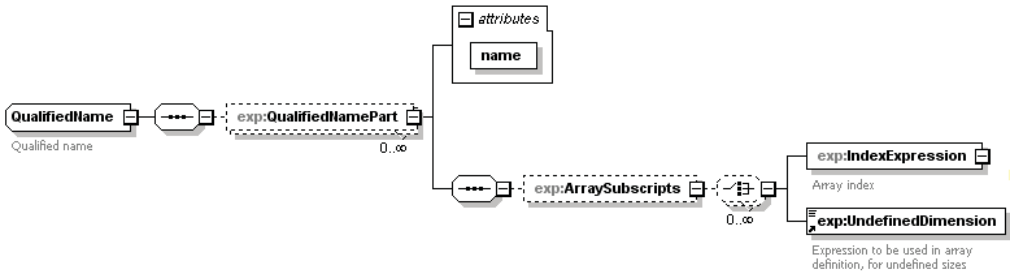
fixed	Defines the meaning of attribute "start". This attribute is only allowed if "start" is also present: • = true: "start" is an initial value of a variable. This is the default. • = false: "start" is a guess value. The variable is used as iteration variable during initialization. After initialization, the variable can have a different value as “start”.

4.2.3 Qualified names

The proposed representation should be neutral with respect of 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 (detailed discussion in Section 4.2.6) more complex types than scalar variables, such as array and records, are allowed. The index of an array can be a general expression, and representating the array's element by a string, e.g. “x[3*1]”, 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. The complex type “QualifiedName” is then defined as in Figure 4.5 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. “QualifiedNamePart” holds a string attribute “name”

Figure 4.5: QualifiedName complex type



and an optional element "ArraySubscripts", to represent the indeces 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 "UndefinedIndex". Conventionally, the first element of an array has index 1. In our representation, array variable definitions are allowed in user-defined functions only.

Example 1. Representation of a qualified name.

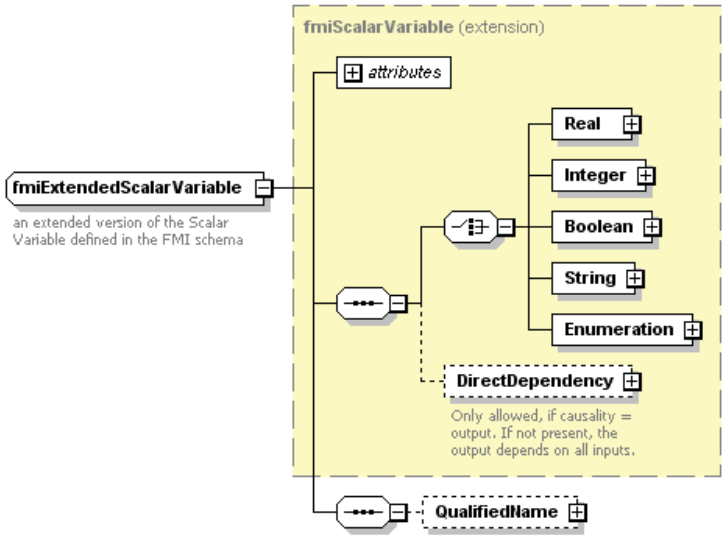
Given a record variable R, and its field x of array type, we would like to represent the name that refers to the second element of the array x in the record R (e.g., "R.x[2]" in Modelica language). The resulting XML representation valid with respect of the XML schema is:

```
<exp:QualifiedName>
        <exp:QualifiedNamePart name="R"/>
        <exp:QualifiedNamePart name="x">
                <exp:ArraySubscripts>
                        <exp:IndexExpression>
                                <exp:IntegerLiteral>2
                                </exp:IntegerLiteral>
                        </exp:IndexExpression>
                </exp:ArraySubscripts>
```

```
        </exp:QualifiedNamePart>
</exp:QualifiedName>
```

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: "fmiScalarVariable" complex type is therefore extended to "fmiExtendedScalarVariable" complex type, that will be the only one used in the final schema, including a "QualifiedName" element as in Figure 4.6.

Figure 4.6: fmiExtendedScalarVariable



4.2.4 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 arithmetical and logical operators; trigonometric, exponential, logarithmic, hyperbolic and other mathematical functions; function calls referring to user-defined functions; variable identifiers and literals.

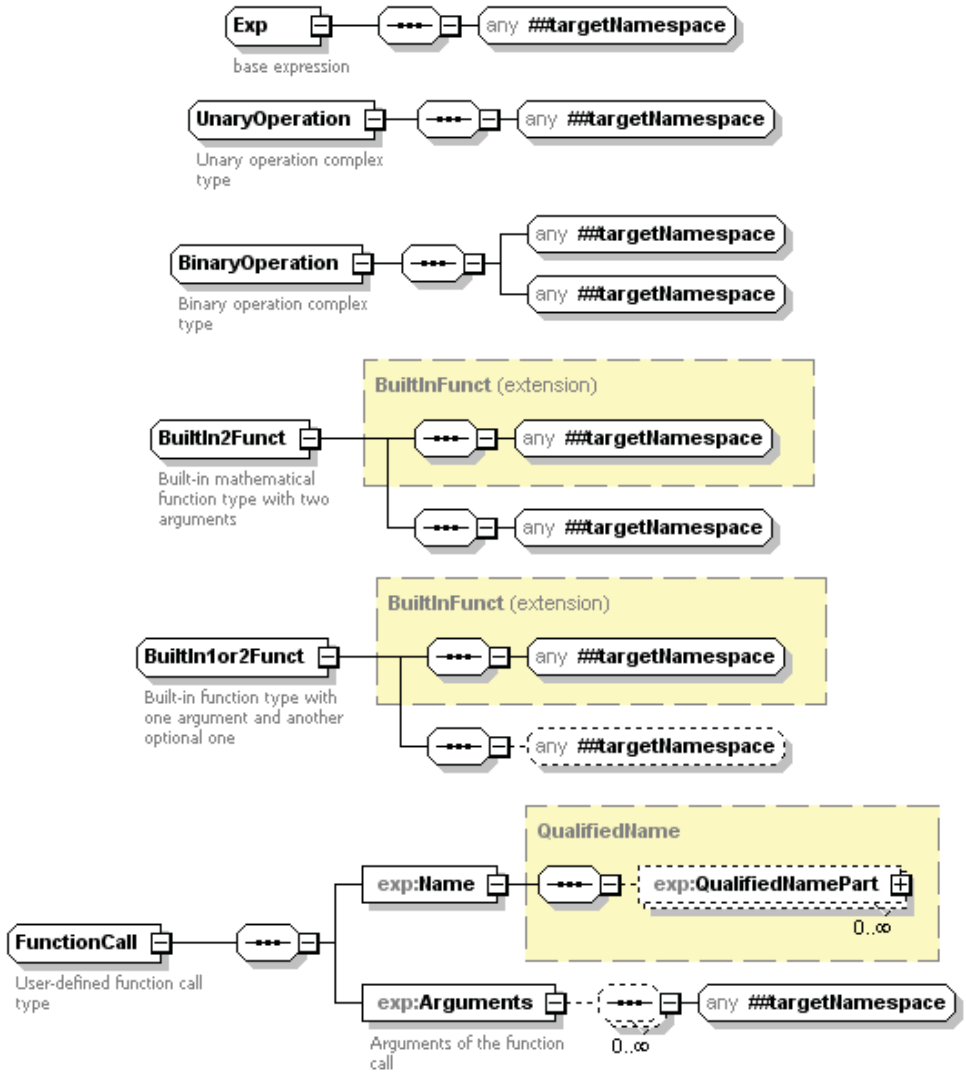
Whenever a valid element is supposed to be a general expression, a wild-card element on the "exp" namespace is used, in order to simplify the schema's extensibility. As a result, when a new expression is needed, it is sufficient to create a new element in the "exp" vocabulary and it will be automatically available in all the rest of the schema.

The "exp" namespace includes the following complex types (Figure 4.7):

complex type	*description*
Exp	A general expression. An element defined as "Exp" type can have any expression defined in the "exp" namespace as possible value
UnaryOperation	Operation having one operand
BinaryOperation	Operation having two operands
BuiltInFunct	Predefined function that requires only one argument as input
BuiltIn2Funct	Predefined function that requires two arguments as input
BuiltIn1or2Funct	Predefined function that can accept either one or two arguments as input
FunctionCall	Function call referring to user-defined function. The function should return one output argument. The arguments should match in type and order the called function interface.
Array	Constructor of an array data structure
RecordConstructor	Constructor of a record data structure
QualifiedName	Structured name representation (see Section 4.2.3)

Table 4.8: "exp" namespace complex types

Figure 4.7: Expression complex types



The following table collect all the allowed basic expressions elements, with their corresponding definition:

Table 4.9: Expression elements

elements	*Type*	*Description*
Add, Sub, Mul, Div,Pow	BinaryOperation	Basic algebraic operations: addition, subtraction, multiplication, division, exponentiation
Neg	UnaryOperation	Negation operand for numeric argument
And, Or	BinaryOperation	Logical operations conjunction and disjunction
Not	UnaryOperation	Logical operation negation. The argument should be a boolean expression.
LogLt, LogLeq, LogGt, LogGeq LogEq, LogNeq	BinaryOperation	Basic logical comparison operations: less than, less or equal than, greater than, greater or equal than, equal than, not equal than. The arguments should be boolean expressions.
Der	BuiltInFunct	Derivative function
Sin, Cos, Tan, Asin, Acos,Atan	BuiltInFunct	Basic trigonometrical functions
Sinh, Cosh, Tanh	BuiltInFunct	Basic hyperbolic functions
Exp, Log, Log10	BuiltInFunct	Exponential and logarithmic functions
Abs	BuiltInFunct	Absolute value function. Given a scalar expression argument x, it returns x if x>=0 or -x if x<0

Sign	BuiltInFunct	Returns 1 if the argument is a positive expression, -1 if the argument is a negative expression
Sqrt	BuiltInFunct	Square root of the argument. The argument is supposed to be a positive real expression.
Atan2	BuiltIn2Funct	four quadrant inverse tangent function
Min, Max	BuiltIn1or2Funct	If the arguments are two scalar expression, the minimum and maximum expression are respectively returned. If the argument is one array of scalar expressions, the minimum and maximum expression are respectively returned
FunctionCall	FunctionCall	Function call referring to user-defined function. The function should return one output argument.
Identifier	QualifiedName	Variable identifier
RealLiteral	real	Real literal expression
IntegerLiteral	integer	Integer literal expression
BooleanLiteral	boolean	Boolean literal expression
StringLiteral	String	String literal expression
Time	none	Time variable

Example 2. Expressions representation

Given the expression $der(x) + 3.5 * sin(x)^2$, an XML representation that is valid according to the defined schema is:

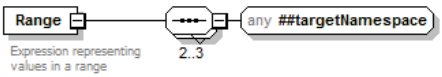
```
<exp:Add>
        <exp:Der>
                <exp:Identifier>
```

```
                    <exp:QualifiedNamePart name='x'/>
                </exp:Identifier>
            </exp:Der>
            <exp:Mul>
                <exp:RealLiteral>3.5</exp:RealLiteral>
                <exp:Pow>
                    <exp:Sin>
                        <exp:Identifier>
                            <exp:QualifiedNamePart name='x'/>
                        </exp:Identifier>
                    </exp:Sin>
                    <exp:IntegerLiteral>2</exp:IntegerLiteral>
                </exp:Pow>
            </exp:Mul>
</exp:Add>
```

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 loops definition, inside algorithms of user-defined functions or as an argument of array constructors. A "Range" element is composed by either two or three scalar expressions. In case there are two expressions, they are considered to be the lower and upper bounds of the range, while by default the step of the loop is 1. If there are three expressions in the "Range" element, then the second one defines the step size of the loop.

Figure 4.8: Range expression



Array variable definitions and uses are allowed only in user-defined func-

tions (Section 4.2.6). The element "UndefinedDimension" can be used in array variables 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.

Figure 4.9: Array element definition

As for arrays, also record variables can be defined and used only in user-defined functions. A "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. The detailed explanation of how to use "Array" and "RecordConstructor" in the left hand side of a user-defined function call equations is postponed to Section 4.2.6.

Figure 4.10: RecordConstructor definition

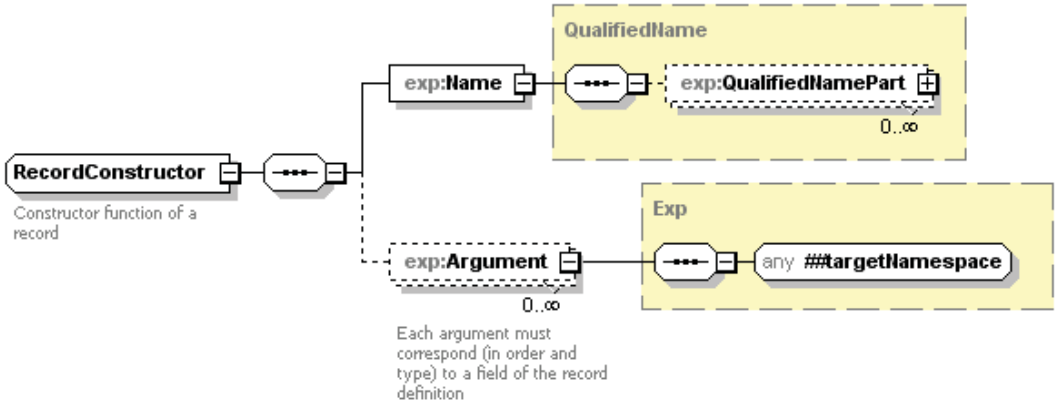
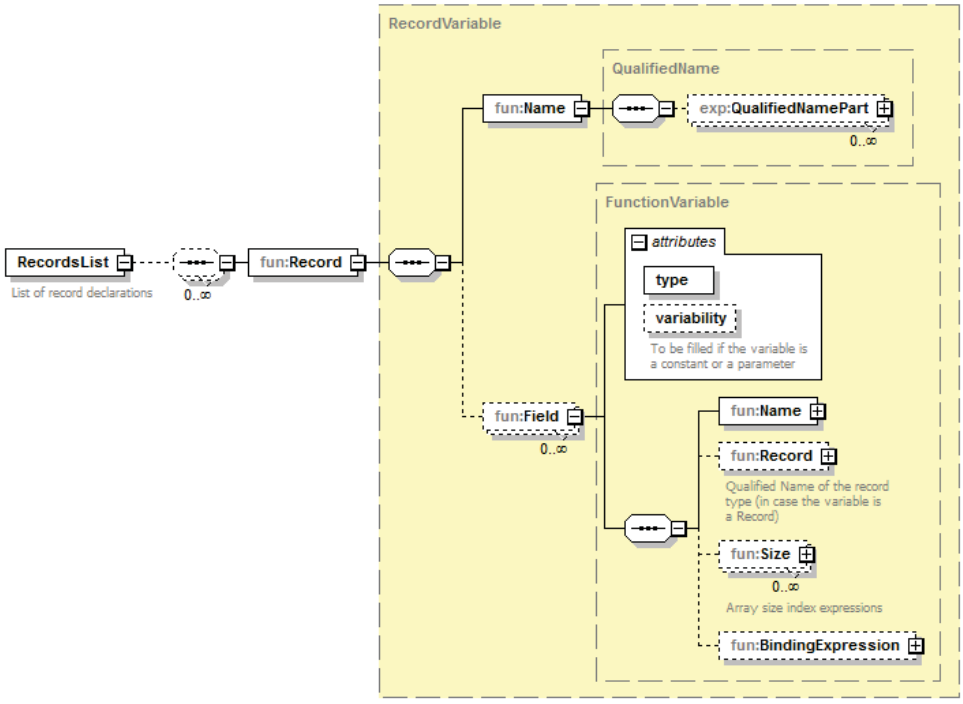


Figure 4.11: RecordList and records definition



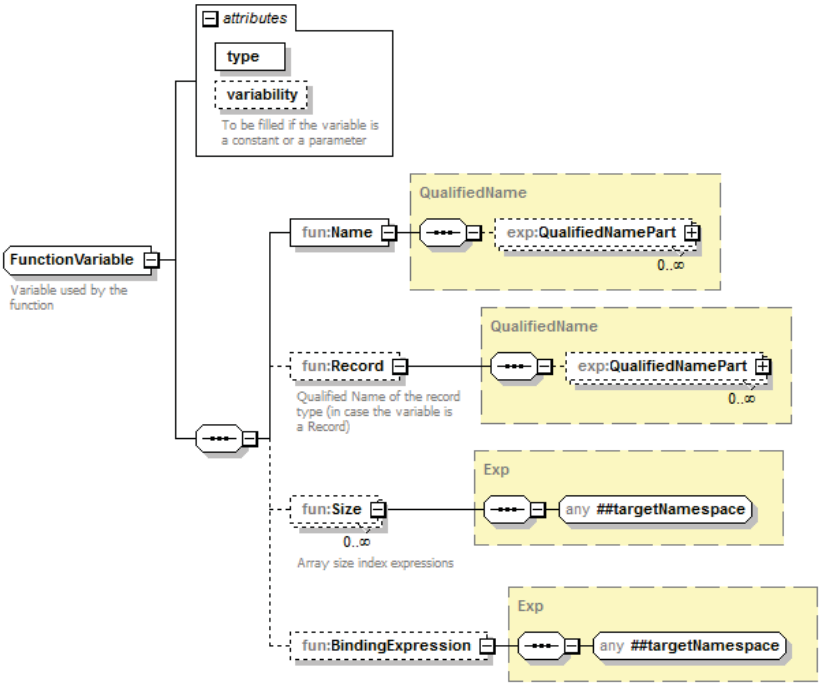
4.2.5 Records definition

A record (also called tuple or struct) is one of the simplest data structures, consisting of two or more values or variables stored in consecutive memory positions, so that each component (called a field or member of the record) can be accessed by applying different offsets to the starting address.

All the elements and complex types relevant to record definitions are stored in the "fun" namespace, since they are mostly related with the use of functions. 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 "RecordsList" element, that is referenced in the main schema (Figure 4.1), should contain the definition of all the records used in the XML document, each one stored in a different "Record" element.

The “Record” elements are declared as “RecordVariable” complex type (as in Figure 4.11) and are defined by a “Name” element and by a list of “Field” elements, one for each field of the record. “Field” elements are declared as “FunctionVariable” complex type, so they hold two attributes: “type” and “variability”, which both can accept a value in a given enumeration. The “type” attribute can assume values within “Real”, “Integer”, “Boolean”, “String” and “Record”. The attribute “variability” is optional and should be used if the given variable is defined as a constant or a parameter. Valid values for “variability” are “constant”, “parameter” and “continuous” (default). In addition to the mandatory element “Name” (of type “QualifiedName”) there are three optional elements: “Record”, “Size” and “BindingExpression”. A “Record” element should be used when “type=’Record”’ and in this case should include the qualified name of the variable record type. If the variable is an array, the element “Size” should include a list of expressions, one for each size defining the variable, possibly of “UndefinedDimension” expression. The “BindingExpression” element defines the initial value of the defined variable.

Figure 4.12: FunctionVariable complex type



Example 3. Records definition

In a model two different types of records are used: "ComplexNumber" and "A". Their definition, in Modelica language is:

```
record ComplexNumber
        Real im;
        Real re;
end ComplexNumber;

record A
        ComplexNumber c;
        constant Real pi = 3.14;
        Real x[2] = {1.2,5.5};
end A;
```

According to the defined XML schema, a valid representation is:

```
<fun:RecordsList>
        <fun:Record>
                <fun:Name>
                        <exp:QualifiedNamePart name='ComplexNumber'>
                </fun:Name>
                <fun:Field type='Real'>
                        <fun:Name>
                                <exp:QualifiedNamePart name='im'>
                        </fun:Name>
                </fun:Field>
                <fun:Field type='Real'>
                        <fun:Name>
                                <exp:QualifiedNamePart name='re'>
                        </fun:Name>
                </fun:Field>
        </fun:Record>

        <fun:Record>
                <fun:Name>
                        <exp:QualifiedNamePart name='A'>
                </fun:Name>
                <fun:Field type='Record'>
                        <fun:Name>
                                <exp:QualifiedNamePart name='c'>
                        </fun:Name>
                        <fun:Record>
                                <exp:QualifiedNamePart name='ComplexNumber'>
                        </fun:Record>
                </fun:Field>
                <fun:Field type='Real' variability='constant'>
                        <fun:Name>
                                <exp:QualifiedNamePart name='pi'>
                        </fun:Name>
                        <fun:BindingExpression>
                                <exp:RealLiteral>3.14</exp:RealLiteral>
```

```
                </fun:BindingExpression>
            </fun:Field>
            <fun:Field type='Real'>
                <fun:Name>
                    <exp:QualifiedNamePart name='x'>
                </fun:Name>
                <fun:Size>
                    <exp:IntegerLiteral>2</exp:IntegerLiteral>
                </fun:Size>
                <fun:BindingExpression>
                    <exp:Array>
                        <exp:RealLiteral>1.2</exp:RealLiteral>
                        <exp:RealLiteral>5.5</exp:RealLiteral>
                    </exp:Array>
                </fun:BindingExpression>
            </fun:Field>
        </fun:Record>
</fun:RecordsList>
```

4.2.6 Functions

A function is a portion of code which performs a specific computation and is relatively independent from 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.

Differently from the variables used in equations, input, output and protected variables of a function can be scalars, but also arrays or records. In this way algorithms can keep the original structure of the variables.

Whereas in the formulation of the equations defined in 4.1 only scalar variables are involved, a detailed discussion on the use of calls for any possible cases in which the function is not scalar is required.

Function calls with non-scalar inputs

If an input of a function is not a scalar, its call will be represented by keeping its structure, possibly using array or record constructors, but populating it with its scalar elements, which are the (scalar) variables of the DAE model. 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, performing the computation required by the function.

Example 4. Function calls with non-scalar inputs

Given the following definition of a record R and a function F:

```
record R
        Real X;
        Real Y[3];
end R;

function F
        Input R X;
        Output Real Y;
end F;
```

A correct function call for F in an equation is the following

```
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 a R record type, and {var1, var2,...,varN} represents an array constructor.

Function calls with 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 unnecessarily duplicated function calls.

Example 5. Function calls with non-scalar output

Considering the following definition of the function f

```
function f
        Input Real X;
        Output Real Y[3];
end f;
```

then equation $x + f(y) * f(z) = 0$ (a scalar product) is mapped into:

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

where x 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 should be treated as all the other scalar variables, thus also their definition is required as explained in Section 4.2.2.

Function calls with multiple outputs

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

$$(out1, out2, ..., outN) = f(in1, in2, ...inM) \tag{4.5}$$

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 even to put the equation in residual form.

Example 6. Function calls with multiple outputs

Given the following definition of a record type R1 and a function F1:

```
record R1
        Real X;
        Real Y[2,2];
end R1;

function F1
        input Real x;
        output Real y[2];
        output R r;
end F1;
```

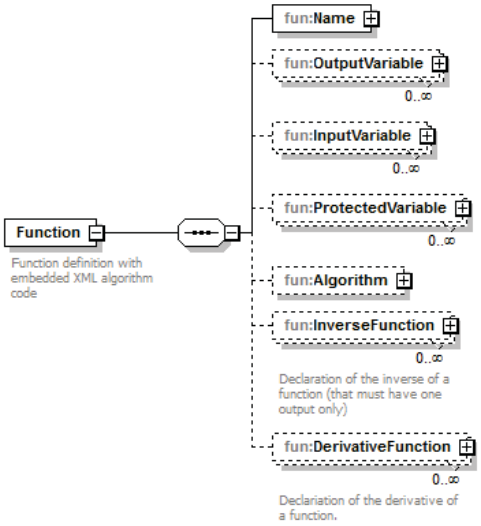
a correct call for the function F1 can be

```
({var1, var2}, R1(var3,{ {var4,var5}, {var6,var7} })) = 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 sizes of the inputs. This solution would lead to less efficient implementations in most target languages.

Figure 4.13: Function definition



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", that 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 as "FunctionVariable" complex type (Figure 4.12).

Example 7. Functions interface definition

Given two complex numbers according to the definition of "ComplexNumber" record as in example 3, a function "getGreatestReal" compares the real part of the two numbers and returns the greatest one, as in the following Modelica language definition:

```
function getGreatestReal
        input ComplexNumber c1;
        input ComplexNumber c2;
        output Real y;
algorithm
        if(c1.re>=c2.re) then
                y:=c1.re;
        else
                y:=c2.re;
        return;
end getGreatestReal;
```

According to the defined schema, a valid XML representation for "getGreatestReal" function (without algorithm definition) is:

```
<fun:Function>
        <fun:Name>
                <exp:QualifiedNamePart name='mulComplexNumbers'/>
        </fun:Name>
        <fun:OutputVariable type='Real'>
                <fun:Name>
                        <exp:QualifiedNamePart name='result'/>
                </fun:Name>
        </fun:OutputVariable>
        <fun:InputVariable type='Record'>
                <fun:Name>
                        <exp:QualifiedNamePart name='c1'/>
                </fun:Name>
                <fun:Record>
                        <exp:QualifiedNamePart name='ComplexNumber'>
                </fun:Record>
        </fun:InputVariable>
        <fun:InputVariable type='Record'>
                <fun:Name>
                        <exp:QualifiedNamePart name='c2'/>
                </fun:Name>
```

```
            <fun:Record>
                <exp:QualifiedNamePart name='ComplexNumber'>
            </fun:Record>
        </fun:InputVariable>

        <fun:Algorithm>
            ... see next example ...
        </fun:Algorithm>

</fun:Function>
```

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.

Every function with only one output argument may have one or more inverse function definitions. The XML structure of the "InverseFunction" element is very similar to the one of "Function" element with a few restrictions: the only output variable of the inverse should be declared as an input of the original function and the output of the original function should be an input of the inverse function. The order of the arguments in the function definition could be permuted.

The definition of "DerivativeFunction" element is slighty extended from the one of "Function", adding an optional integer attribute "order" (default is 1) and two new optional boolean attributes, "derived" and "zeroDerivative" to the input variable definition. The derivative function should have at least one output and one real input. An input variable has "derived=true" if the function is differentiated with respect of the variable. Only real input variables can have "derived=true". The function should be differentiated with respect of the variables in order (from the first to the last). At least one input must be real and have "derived=true". The derivative function is only valid if variables with "zeroDerivative=true" are independent of the variables

the function is differentiated with respect to (i.e. the derivative of the input variable is "zero").

The elements and complex types useful for the description of the algorithm are defined in a different schema module than the “Function” element, but always under the “fun” namespace. This module includes the complex types:

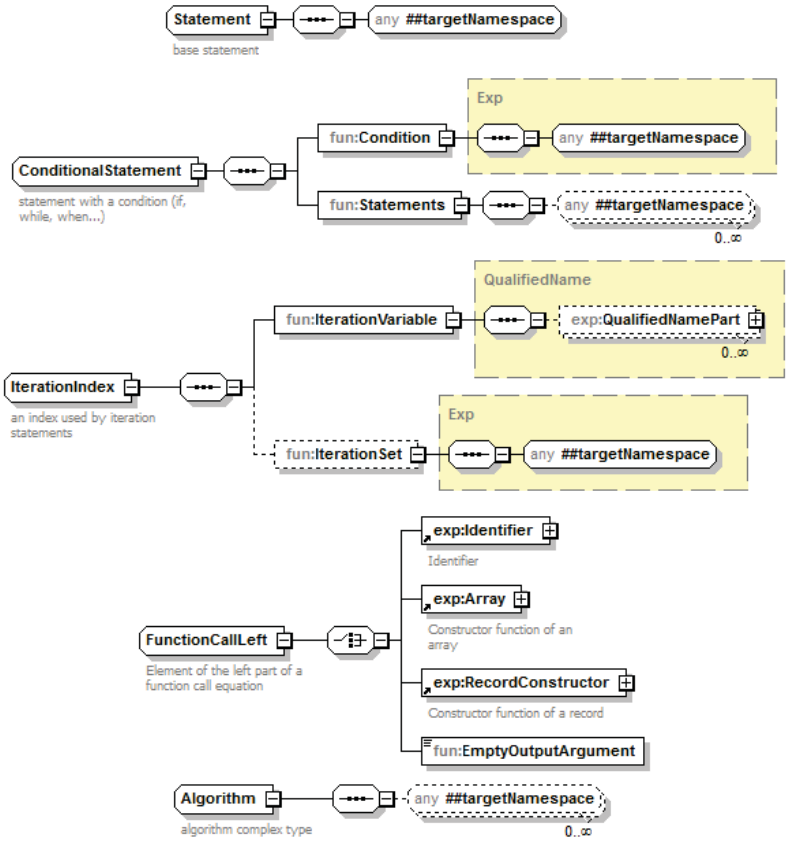
complex type	*description*
Statement	Wildcard that can be matched with any possible statement defined in the schema
ConditionalStatement	A conditional statement, defined by a boolean condition expression and a list of statements that should be executed if the condition is satisfied.
IterationIndex	Index for use in loop definitions. It requires a variable to iterate inside a set of possible values. This set can either be a Range or an Array expression.
FunctionCallLeft	It represents the left hand side of a function call statement. “FunctionCallLeft” can be an identifier of variable, an array or record constructor populated with variable identifiers, empty arguments or any possible combination of these elements. The FunctionCallLeft definition should match the output variables of the called function, in order and type.
Algorithm	An algorithm is defined as a list of statements, possibly empty.

Table 4.12: “fun” namespace complex types

An algorithm is defined by a list of statements, possibly empty. A statement can be an assignment, a conditional statement or a loop statement.

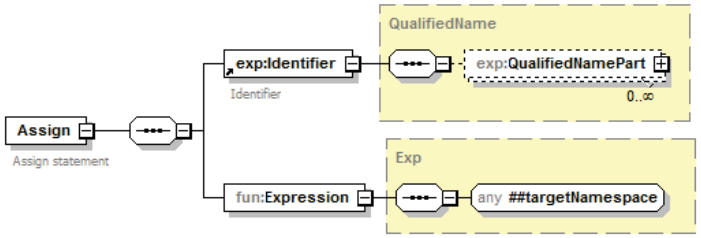
An assignment is represented by the element “Assign”, that has two children element: the first one should be an identifier of variable, while the second one is the expression that represents the new assigned value to the

Figure 4.14: Complex types for algorithms representation



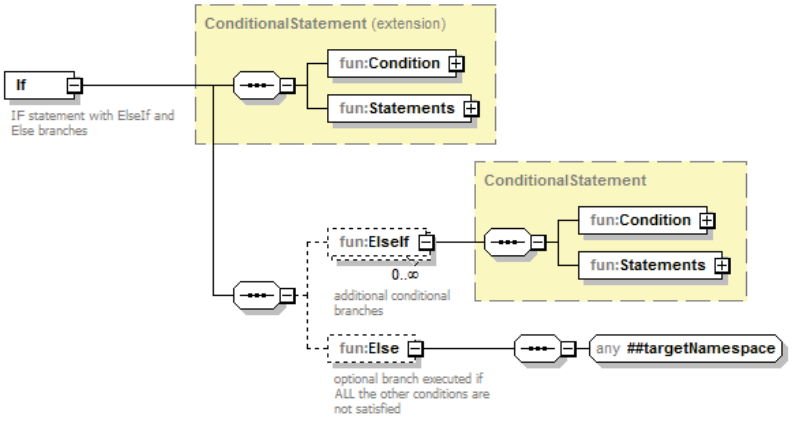
given variable.

Figure 4.15: Assign element



The "If" element is designed as an extension of "ConditionalStatement". In fact, in addition to the first conditional branch, the "If" element provides optional "Else" elements, defined as "ConditionalStatement" complex type, and "ElseIf" branch. In case the condition of the "If" element is not satisfied, "Else" elements are taken into account in document order: when a condition of an "Else" element is satisfied, then the corresponding statements should be executed. If none of the conditions both of "If" element and of any "Else" element is satisfied, then the statements defined as children of the "ElseIf" element should be executed.

Figure 4.16: If statement



Two possible loops elements are defined in the schema: "While" for con-

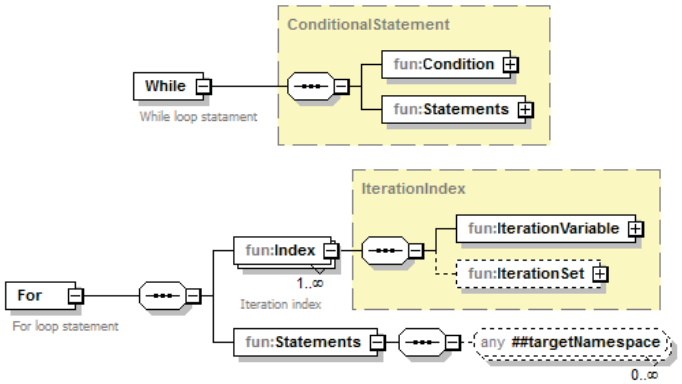
ditional loops and "For" for iteration loops.

The "While" element is defined as "ConditionalStatement" complex type, so it has a condition and a list of statements to be executed if the condition is satisfied.

The "For" statement is defined by an iteration index and a list of statements. The iteration index (element "Index") defines in which set ("IterationSet") a given variable ("IterationVariable") should iterate. The "IterationSet" expression should be an "Array" or a "Range" element (see Section 4.2.4) and the list of statements should be executed for every value of "IterationVariable" within the "IterationSet". The iteration variable should be of the same type as the type of the elements in the iteration set and it should not be assigned inside by the statements of the loop.

The empty "Break" element can only be used in the statement list of a "While" or "For" element in order to ask to end the execution of a loop at a given point.

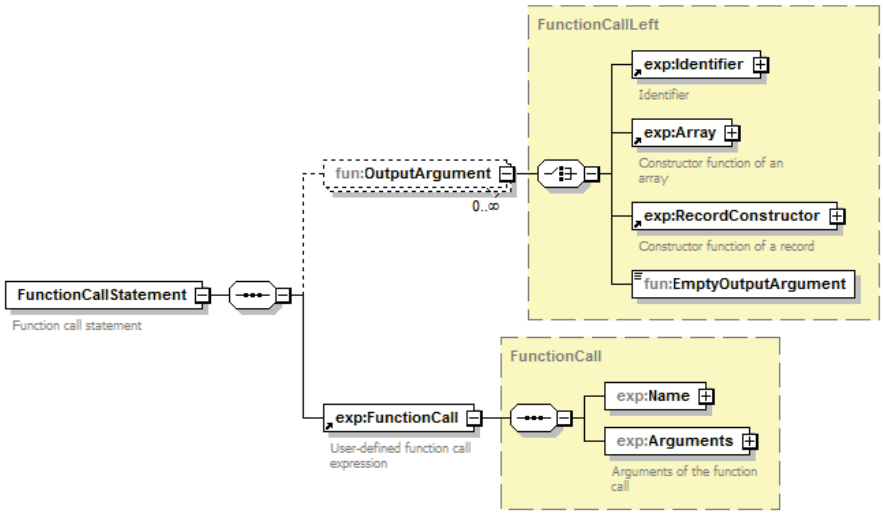
Figure 4.17: Loops definition: While and For elements



"FunctionCallStatement" defines a special assignment where the right hand side is a call to an user-defined function, possibly returning complex arguments. The left hand side contains the output arguments, i.e. the variables that accept the values returned from the function. The variables of the left

hand side of "FunctionCallStatement" can be variable identifiers, constructor of arrays and records filled with variable identifiers or empty arguments, represented by the "EmptyOutputArgument" element. The variables in the left hand side of the function call statement should be compatible, in order and type, with the values returned by the function of the right hand side. The "EmptyOutputArgument" placed in the left hand side means that no assignment is needed for the returned value of the function at corresponding position.

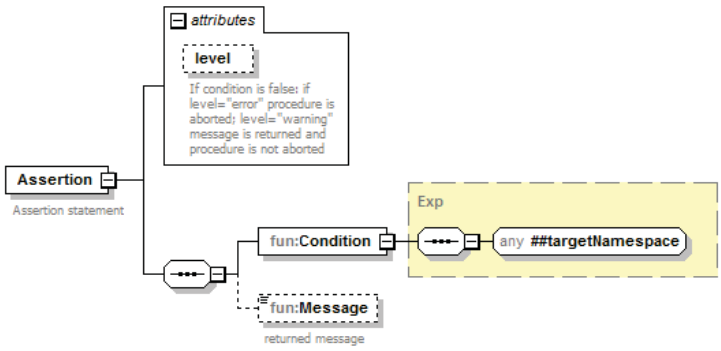
Figure 4.18: FunctionCallStatement definition



"Return" element indicates that the execution of the algorithm should be ended at the given point and the output variables should be returned with their current values.

To conclude, in the algorithms is possible to represent assertions using the "Assertion" statement element. An assertion is defined by an optional "Message" string element, a "Condition" element that should be a boolean expression and an optional attribute "level" that can assumes value "error" (default value) or "warning". If the condition is false, then nothing happens, else if the condition is satisfied then the message should be returned and the

Figure 4.19: Assertion element



execution aborted in case "level=error".

Example 8. Algorithms representation

In order to complete the representation of "getGreatestReal" defined in example 7, the algorithm representation should be included.

The corresponding XML mapping is:

```
<fun:Algorithm>
        <fun:If>
                <fun:Condition>
                        <exp:LogGeq>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='c1'>
                                        <exp:QualifiedNamePart name='re'>
                                </exp:Identifier>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='c2'>
                                        <exp:QualifiedNamePart name='re'>
                                </exp:Identifier>
                        </exp:LogGeq>
                </fun:Condition>
                <fun:Statements>
                        <fun:Assign>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='result'>
```

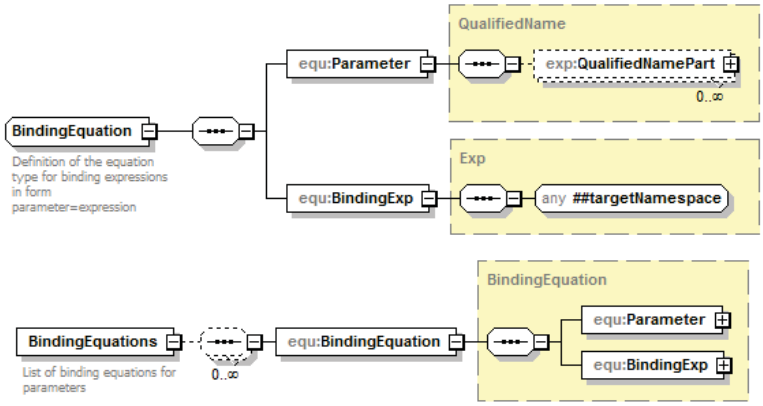
```
                                </exp:Identifier>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='c1'>
                                        <exp:QualifiedNamePart name='re'>
                                </exp:Identifier>
                        </fun:Assign>
                </fun:Statements>
                <fun:Else>
                        <fun:Assign>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='result'>
                                </exp:Identifier>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name='c2'>
                                        <exp:QualifiedNamePart name='re'>
                                </exp:Identifier>
                        </fun:Assign>
                </fun:Else>
        </fun:If>
        <fun:Return/>
</fun:Algorithm>
```

4.2.7 Equations

Complex types and elements regarding the equations of the DAE system are collected under the "equ" namespace.

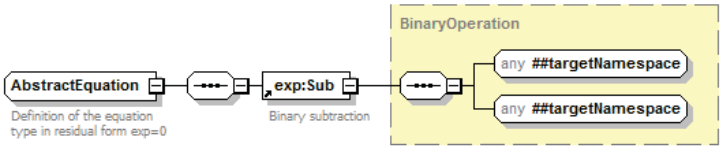
Once defined the expressions, mapping the mathematical formulation of the binding equations (4.3) to the XML schema is straightforward. In the "equ" namespace a complex type "BindingEquation" is defined. It provides an element "Parameter" of "QualifiedName" type that represents the left hand side of 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, as in Figure 4.20.

Figure 4.20: Binding Equations



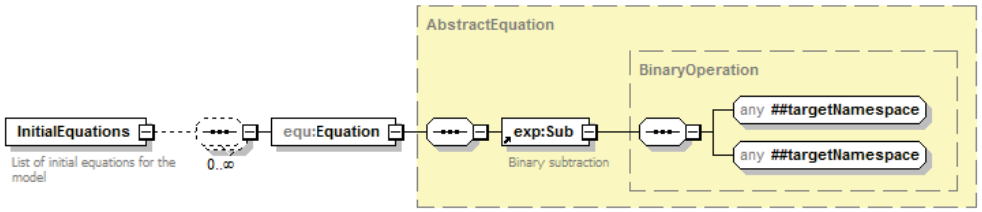
Equations in residual form are represented by the complex type "AbstractEquation". This type of equations provide a subtraction node to represents an equation in $exp1 - exp2 = 0$ form.

Figure 4.21: AbstractEquation complex type



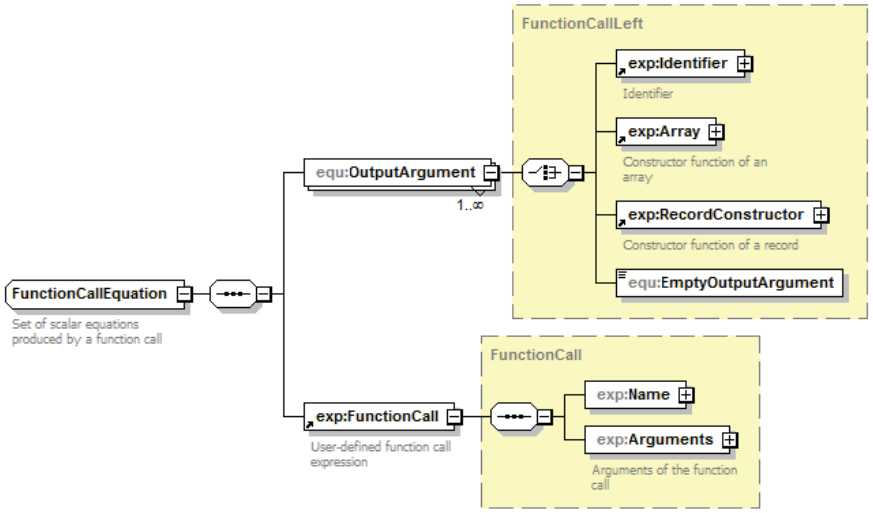
The initial equations set (4.4) is represented by the element "InitialEquations", that collects a list, possibly empty, of "Equation" elements defined as "AbstractEquation" complex type.

Figure 4.22: Initial Equations



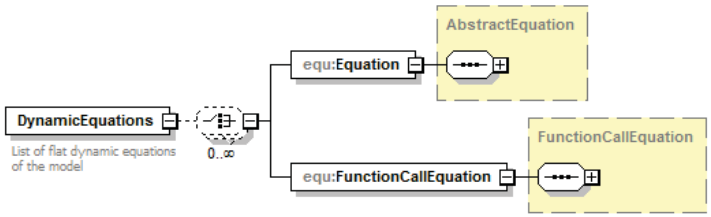
The set of dynamic equations (4.2) is mapped to the "DynamicEquations" element. According to the considerations expressed in paragraph 4.2.6, equations resulting from the call of functions with multiple outputs are not suitable for a representation in residual form. Thus a complex type for mapping the equation form (4.5) is given by the complex type "FunctionCallEquation". The left hand side of the equation (4.5) is represented by the a set of "OutputArgument" elements, defined as "FunctionCallLeft" complex type, that can have as children scalar variable identifiers, array or record constructors populated with scalar variables identifiers, empty arguments, or any possible combination of these elements. The right hand side is a "FunctionCall" (see 4.2.4). 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).

Figure 4.23: FunctionCallEquation complex type



Hence, the "DynamicEquations" elements contains a list of "Equation" elements of "AbstractEquation" type, which represent equations in residual form, and "FunctionCallEquation" elements, which represent the equations (4.5).

Figure 4.24: DynamicEquations element



The elements "BindingEquations", "DynamicEquations" and "InitialEquations" are directly referenced and used in the main schema (see Figure4.1).

4.3 Possible applications

Depending on the specific application, different subsets of the equation system (4.2)-(4.4) are needed.

For off-line simulation, the parameters and constants within each equation are firstly numerically evaluated, for example by solving all the three equation subsets (4.2)-(4.4) together or after ordering the equations by the Tarjan's algorithm [32], to determine all the initial values. Once fixed to the numerical values of each parameter, the dynamic equations can be used to compute derivatives and algebraic variables at each step of the integration algorithm.

Real-time simulation code can obtained by suitable transformation of the set (4.2) only, e.g. by in-lining the discretization method within the equations of the XML file using forward or backward Euler's method.

LFT is a widely used description formalism used in control and system identification theory. The transformation of dynamic models with uncertain parameters and nonlinearities into LFT formalism requires the system (4.2)-(4.3) to be considered as a whole. The procedure for obtaining the LFT representation can be found on [12].

In robotics, subsets of (4.2)-(4.3) can be suitably solved in order to obtain direct/inverse kinematics, computed torque controllers, and inverse dynamics controllers.

More advanced applications would require to linearize the equations (4.2)-(4.3) to obtain the generalize impedance of the structure in certain configurations, possibly representing it as an LPV/LFT system.

Model order reduction algorithms would first require to solve one or more simulation problems to obtain reference scenario(s), and then apply symbolic/numeric approximations to (4.2) in order to obtain a more compact and simple model, which is able to replicate the reference scenario(s) within specified error bounds. For parameter sensitivity studies, this procedure could be applied to the set (4.2)-(4.3).

4.4 An extension example: DAE optimization problem

4.4.1 Mathematical formulation of optimization problems

Consider the following formulation of a dynamic optimization problem:

$$min_{u(t),p}\, \Psi(x(t_i), y(t_i), u(t_i), p),\ i \in 1 \ldots N_{cost} \tag{4.6}$$

subject to the dynamic of the system

$$F_i(x(t), \dot{x}(t), u(t), v(t), p, t) = 0,\ t \in [t_0, t_f] \tag{4.7}$$

and the constraints

$$c_{ineq}(x(t), y(t), u(t), p) \leq 0\ t \in [t_0, t_f] \tag{4.8}$$

$$c_{eq}(x(t), y(t), u(t), p) = 0\ t \in [t_0, t_f] \tag{4.9}$$

$$c^p_{ineq}(x(t_j), y(t_j), u(t_j), p) \leq 0\ j \in 1...N_{ineq},\ t_j \in [t_0, t_f] \tag{4.10}$$

$$c^p_{eq}(x(t_k), y(t_k), u(t_k), p) = 0\ j \in 1...N_{eq},\ t_k \in [t_0, t_f] \tag{4.11}$$

where $x(t) \in \mathbb{R}^{n_x}$ are the dynamic variables, $y(t) \in \mathbb{R}^{n_y}$are the algebraic variables, $u(t) \in \mathbb{R}^{n_u}$are the control inputs, and $p \in \mathbb{R}^{n_p}$ are parameters which are free in the optimization. In addition, the optimization is performed on the interval $t \in [t_0, t_f]$, where t_0 and t_f can be fixed or free, respectively. In addition, the initial values of the dynamic and algebraic variables may be fixed or free in the optimization. The constraints include inequality and equality path constraints, (4.8)-(4.9). In addition, inequality and equality

point constraints, (4.10)-(4.11), 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. The cost function (4.6) is a generalization of a terminal cost function, $f(t_f)$, in that it admits inclusion of variable values at other time instants. This form includes some of the most commonly used cost function formulations. Obviously, terminal as well as initial costs are included. A Lagrange cost function can be obtained by introducing an additional state variable, $x_L(t)$, with the associated differential equation $\dot{x_L}(t) = L(x(t), u(t))$, and the cost function $\Psi(t_f) = x_L(t_f)$. The need to include variable values at discrete points in the interior of the optimization interval in the cost function arises for example in parameter estimation problems. In such cases, a sequence of measurements, $y_d(t_i)$, obtained at the sampling instants t_i, $i \in 1 \ldots N_d$ is typically available.

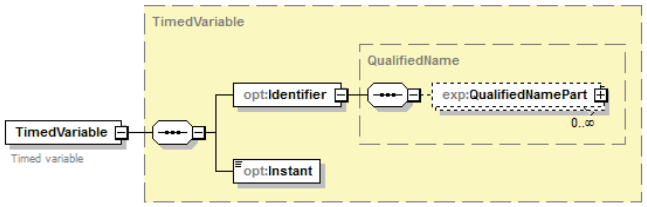
4.4.2 XML schema extension

The extension of the schema for the representation of optimization problems has been designed in a different module having namespace "opt".

The first step in the design of the extension has been to extend the set of possible expressions, including the definition of timed variable. The "TimedVariable" element is composed by two elements: "Identifier", defined as qualified name, and "Instant", that should hold a numerical value. The "Instant" value indicates that the "TimedVariable" element refers to the value of the variable in the given time instant. The value of "Instant" should be included in the optimization time interval.

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 include inequality and equality path constraints, but also point constraints are supported. Point constraints are typically used to express initial or ter-

Figure 4.25: TimedVariable element definition



minal constraints, but can also be used to specify constraints for time points in the interior of the interval.

The "ObjectiveFunction" element is represented by an expression. The elements "IntervalStartTime" and "IntervalFinalTime" represent boundary values of the optimization interval. These two elements are defined as "TimeVariable" complex type, which allows to define if the value is either free or not, using the boolean element "Free" (default "false") and a value, in case the variable is fixed, or optionally an initial guess if the variable is free. The "TimePoints" element allows to define a list of point constraints, with relative index and value. Finally the element "Constraints" contains a list of constraints, which can be of equality ("ConstraintEq"), or inequality ("ConstraintGeq" for greater-equal constraints and "ConstraintLeq" for less-equal constraints). Each constraint element accepts as children two expressions, on which the constraint is applied. An overall of the extension is given in Figure 4.26.

The "opt" namespace is then imported in the main DAE XML schema and the root element "Optimization" is directly referenced, as in Figure 4.27.

Figure 4.26: Optimization problem extension

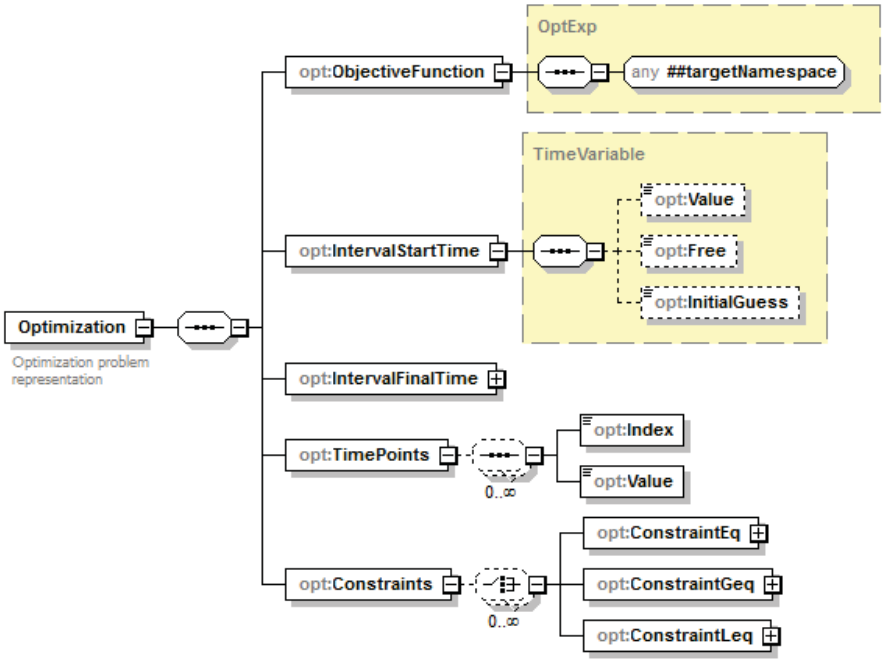


Figure 4.27: Overall of the schema extended by the optimization module

Chapter 5

XML representation of DAE systems obtained by continuous-time Modelica models

5.1 Preliminary handling of the model

The proposed schema allows to represent continuous-time DAE only, so conditional and discrete event models are not currently supported.

Before mapping the Modelica model, it should be firstly flattened. The resulting model should contain the system of equations of every component of the system and the connection equations. Also for-clauses in the equation section of the model (e.g. the loop in the next example model) should be solved in order to obtain the final corresponding set of equations. Every variable involved in the equations can only be a scalar, so more complex data structures needed to be flattened. Qualified names can help to maintain the relation between the flattened scalar variable and the original data structure and original model component. In the same way, scalar functions applied to array arguments should be handled to have the resulting set of scalar functions applied to scalar arguments (see [35, Section 12.4.5]).

Example 9. Given the following simple Modelica model

```
class FiveEquations
        Real[5] x;
equation
        for i in 1:5 loop
                x[i] = i+1;
        end for;
end FiveEquations;
```

an equivalent model suitable for the XML representation is

```
class FiveEquations
        Real x1;
        Real x2;
        Real x3;
        Real x4;
        Real x5;
equation
        x1 = 2;
        x2 = 3;
        x3 = 4;
        x4 = 5;
        x5 = 6;
end FiveEquations;
```

Functions can keep array and record data structures in their definition, while function calls used by equations can use array and record constructors to handle the relation between scalar variables and complex data arguments. Additional scalar variables could be added in the flattened model to manage operations using function calls with one complex data argument. The detailed explanation on how to handle functions has been given in Section 4.2.6.

It is important to notice that the majority of the controls on the correct-

ness of the model, such as type checking and structural analysis, can only be performed by the compiler and should be done before exporting the model in XML format.

Example 10. The following Modelica model contains array variables, a function applied to an array input argument and returning an array output argument and equations performing operations involving the defined function.

```
class Example
        Real[3] u = {1,2,3};
        Real[3] v = {3,4,5};
        Real z;

equation
        z = F(u) * F(v);

function F
        input Real[3] x;
        output Real[3] y;
algorithm
        y := x;
        return;
end F;

end Example;
```

The model should be processed in order to:

- unroll the array variables involved in the equation section, redefining as a set of scalar variables ;
- move the equations in the variable definitions section to the "equation" section, together with all the other dynamic equations. Notice that only binding equations involving parameter with "fixed=true" attribute are not treated as all the other equations by the schema (see Section 4.2.7);

- add scalar auxiliary variables and the relative equations that assign them to the output argument of the function F (see Section 4.2.7);
- make qualified names explicit.

The resulting flattened model is obtained

```
fclass FExample
        Real u[1];
        Real u[2];
        Real u[3];
        Real v[1];
        Real v[2];
        Real v[3];
        Real z;
        Real temp_1[1];
        Real temp_1[2];
        Real temp_1[3];
        Real temp_2[1];
        Real temp_2[2];
        Real temp_2[3];
equation
        u[1] = 1;
        u[2] = 2;
        u[3] = 3;
        v[1] = 3;
        v[2] = 4;
        v[3] = 5;
        ({temp_1[1],temp_1[2],temp_1[3]}) = FExample.F({u[1],u[2],u[3]});
        ({temp_2[1],temp_2[2],temp_2[3]}) = FExample.F({v[1],v[2],v[3]});
 z = (temp_1[1])*(temp_2[1])+(temp_1[2])*(temp_2[2])+(temp_1[3])*(temp_2[3]);



function FExample.F
        input Real[3] x;
        output Real[3] y;
```

```
algorithm
        y[1] := x[1];
        y[2] := x[2];
        y[3] := x[3];
        return;
end FExample.F;

end FExample;
```

5.2 Mapping Modelica models to the XML schema

Once the model has been pre-processed the mapping to schema is conceptually straightforward:

- the variables declared in the first section of the model should match the elements of "ModelVariables" (see Section 4.2.2);
- binding equations involving parameters with attribute "fixed=true" should match the "equ:BindingEquations" element;
- the element "equ:DynamicEquations" represents the "equation" part of the Modelica model. These equations can be represented either in residual form ("equ:Equation" element) or in the form 4.5 ("equ:FunctionCallEquation" element) as explained in Section 4.2.7;
- the equations in the section "initial equation" of the Modelica model should match the "equ:InitialEquations" element definition;
- every function definition should be included in the element "fun:FunctionsList". Annotations on derivative and inverse functions can optionally be used to represents them using "fun:DerivateFunction" and "fun:InverseFunction" (see Section 4.2.6);
- record definitions should match the element "RecordsList".

Figure 5.1: Matching a Modelica model with the XML schema

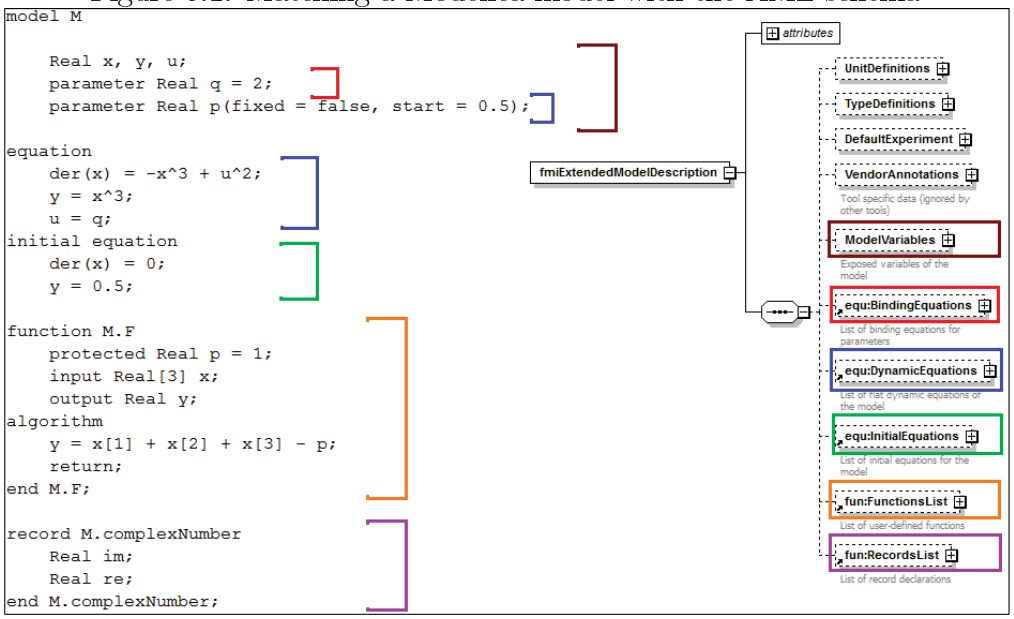


Figure 5.1 graphically shows how every section of the Modelica model is matched to the XML schema sections.

From a practical point of view, the best way to export Modelica models to XML documents valid with respect of the proposed schema is to implement in the compiler a code generation module that traverses the syntax tree of the language, encoding every node with the respective XML representation. In fact the structure of an XML document can easily be mapped to a tree data structure, as well as the syntax of a language. In the next section it is shown how the code generation has been implemented in the JModelica.org compiler.

Chapter 6

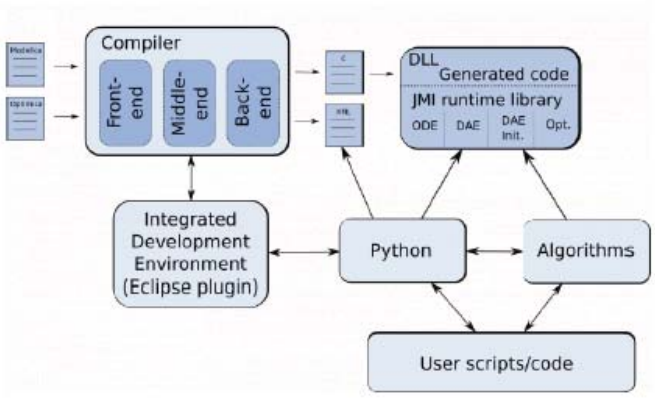
Implementation and test case

6.1 The JModelica.org platform

JModelica.org [27, 4] is an extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems. The main objective of the project is to create an industrially viable open source platform for optimization of Modelica models, while offering a flexible platform serving as a virtual lab for algorithm development and research. As such, JModelica.org is intended to provide a platform for technology transfer where industrially relevant problems can inspire new research and where state of the art algorithms can be propagated form academia into industrial use. JModelica.org is a result of research at the Department of Automatic Control, Lund University, [1] and is now maintained and developed by Modelon AB [26] in collaboration with academia.

A unique feature of JModelica.org is the support for the innovative extension Optimica [2]. Optimica enables to conveniently formulate optimization problems based on Modelica models using simple but powerful constructs for encoding of optimization interval, cost function and constraints. Optimica also features annotations for choosing and tailoring the underlying numerical optimization algorithm to a particular optimization problem.

Figure 6.1: JModelica.org architecture



The JModelica.org compilers are developed in the compiler construction framework JastAdd. JastAdd is based on established concepts, including object orientation, aspect orientation and reference attributed grammars. Compilers developed in JastAdd are specified in terms of declarative attributes and equations which together forms an executable specification of the language semantics. In addition, JastAdd targets extensible compiler development which makes it easy to experiment with language extensions. A full documentation of the JastAdd framework is available in [33].

For user interaction JModelica.org relies on the Python language [30]. Python offers an interactive environment suitable for scripting, development of custom applications and prototype algorithm integration. The Python packages Numpy and Scipy provide support for numerical computation, including matrix and vector operations, basic linear algebra and plotting. The JModelica.org compilers as well as the model executables/dlls integrate seemlessly with Python and Numpy.

An overview of JModelica.org platform in given in [3].

6.2 JModelica.org Abstract Syntax Tree (AST)

A fundamental data structure in most compilers is the Abstract Syntax Tree (AST). An AST serves as an abstract representation of a computer program and is often used in a compiler to perform analyses (e.g., binding names to declarations and checking type correctness of a program) and as a basis for code generation.

Three different ASTs are used in the JModelica.org front-ends.

- The source AST results from parsing of the Modelica or Optimica source code. This AST shares the structure of the source code, and consists of a hierarchy consisting of Java objects corresponding to class and component declarations, equations and algorithms. The source AST can also be used for unparsing, i.e., pretty printing of the source code.

- The instance AST represents a particular model instance. Typically, the user selects a class to instantiate, and the compiler then computes the corresponding instance AST. The instance AST differs from the source AST in that in the former case, all components are expanded down to variables of primitive type. An important feature of the instance AST is that it is used to represent modification environments; merging of modifications takes place in the instance AST. As a consequence, all analysis, such as name and type analysis takes is done based on the instance AST.

- The flat AST represents the flat Modelica model. Once the instance AST has been computed, the flat AST is computed simply by traversing the instance AST and collecting all variables of primitive type, all equations and all algorithms. The flat AST is then used, after some transformations, as a basis for code generation.

For more information on how the JModelica.org compiler transforms these ASTs, see the paper [5].

6.3 Exporting models as XML documents

The XML code generation has been implemented by exploiting the aspect-oriented design allowed by the JastAdd framework. Aspects allow to add features to AST classes without having to syntactically edit those classes. For example, when implementing type checking, it is just needed to add some specific behavior to most of the AST classes. This behavior can be grouped together into an aspect. In the same manner, to extend the system with XML code generation, a new aspect module has been added for the purpose. When the compiler is built, JastAdd waves the methods defined into the aspect modules to the correct Java classes (i.e. to the correct node type of the AST).

JastAdd also supports attributes in the sense of attribute grammars: attributes are declared in AST classes, and their values are defined by equations. As in attribute grammars, an attribute is either synthesized or inherited depending on if it is used for propagating information upwards or downwards in the AST. This feature has been used to define an attribute "ASTNode.xmlTag()" that returns the correct XML for a given tree node with respect of the proposed XML schema.

Example 11. XML tag attributes.

The following code is a portion of the "XMLTagBinding" aspect module and defines the attribute "xmlTag" for the AST nodes involved in the representation of function in the JModelica.org compiler.

```
aspect XMLTagBinding{
        //functions
        syn String FFunctionDecl.xmlTag();
        eq FFunctionDecl.xmlTag() = "Function";
```

```
        syn String FAlgorithmBlock.xmlTag();
        eq FAlgorithmBlock.xmlTag() = "Algorithm";
        syn String FStatement.xmlTag();
        eq FBreakStmt.xmlTag() = "Break";
        eq FReturnStmt.xmlTag() = "Return";
        eq FAssignStmt.xmlTag() = "Assign";
        eq FFunctionCallStmt.xmlTag() = "FunctionCallStatement";
        eq FIfStmt.xmlTag() = "If";
        eq FForStmt.xmlTag() = "For";
        eq FWhileStmt.xmlTag() = "While";
}
```

The keyword "syn" allows to define a synthesized attribute. As an example, "FStatement" is an abstract node of the AST used as a base for the definition of every statement allowed in user-defined function algorithms, and "syn String FStatement.xmlTag()" creates a new attribute for the class, requiring that every node type extending "FStatement" should have a value for the "xmlTag" attribute. Hence, the keyword "eq" is used to define an equation that binds the value of the tag that matches the definition of the XML schema. In the same manner attributes have been defined for every kind of node of the JModelica.org abstract syntax tree.

JModelica.org compiler provides a module "XMLGenerator" which takes a model, described by a tree with "FClass" root element, and uses a template for the static general structure of the XML document. The template has been designed by matching the same structure of the proposed XML schema.

```
<fmiExtendedModelDescription $XML_namespaces$  $XML_rootAttributes$>

$XML_unitDefinitions$
$XML_typeDefinitions$
$XML_defaultExperiment$
$XML_vendorAnnotations$
<ModelVariables>$XML_variables$</ModelVariables>
```

```
$XML_bindingEquations$
$XML_Equations$
$XML_initialEquations$
$XML_Optimization$
$XML_Functions$

</fmiExtendedModelDescription>
```

The value of each portion of the template denoted by "$XML_variable$" name is given in the "XMLGenerator" module.

Example 12. XML code generation for the function definitions section

The following code defines the values of "$XML_Functions$" variable and then it starts the XML code generation for user-defined functions.

```
class DAETag_XML_functions extends DAETag {

public DAETag_XML_functions (AbstractGenerator myGenerator, FClass fclass) {
        super("XML_Functions","Functions Declaration", myGenerator,fclass);
}

public void generate(PrintStream genPrinter) {
        Boolean generateEqu =
                fclass.root().options.getBooleanOption("generate_xml_equations");
        if(generateEqu) {
                genPrinter.println("<fun:FunctionsList>");
                for(FFunctionDecl f : fclass.getFFunctionDeclList())
                        f.prettyPrint_XML(genPrinter,"\t\t",f);
                genPrinter.println("\t</fun:FunctionsList>");
        }
}

}
```

The "generate_xml_equations" boolean attribute can be set by the user and its values is "true" if the user requests the generation of the XML representation for the given model (i.e., "fclass" argument). If this is the case,

the previous code generates the root tag "<fun:FunctionsList>" and calls the "prettyPrint_XML()" method for every function definition in the AST (i.e. "FFunctionDecl" nodes), matching the definition given in Section 4.2.6.

The portions of the XML documents relative to the FMI schema part, variable definitions, equations, optimization problem (for Optimica models only) are generated in the same manner. The definition of records is missing, since the record support was only partially implemented in the JModelica.org platform at the time the present work was being developed.

Once the structure of the XML document has been defined, the last step is to implement a method that given every node in the tree representing a model generates the corresponding XML tags. This method is "ASTNode.prettyPrint_XML()", also used in the previous example, and it is defined for every possible node in the "XMLCodeGen" aspect, trying to exploit in the most efficient way as possible the inheritance of the AST tree definition. e.g., the "prettyPrint_XML()" method has been defined only once for the node type "FMathematicalFunctionCall" node, which is extended by every built-in mathematical function, since the XML representation of these elements has the same structure. The "prettyPrint_XML()" method iteratively calls itself for every children node, in order to traverse the AST and generates the XML representation until the leaves are reached.

Example 13. Qualified names XML code generation.

The following code generates the XML code corresponding to qualified names (see Section 4.2.3). Since the qualified names structure can be seen as a tree, the example is representative of how the XML code is generated for the whole model.

```
public void FQName.prettyPrint_XML(
                Printer p, PrintStream str, String indent, Object o){

                for (FQNamePart np : this.getFQNamePartList())
                        np.prettyPrint_XML(str,p.indent(indent));
                }
```

```
public void FQNamePart.prettyPrint_XML(
                Printer p, PrintStream str, String indent, Object o){
String namespace = "exp";
String tag = this.xmlTag();
String nameAttr = "␣name=\"" + this.getName() + "\"";

        if (this.hasFArraySubscripts()){
                str.println(indent + "<" + namespace + ":" + tag + nameAttr + ">");
                this.getFArraySubscripts().prettyPrint_XML(str,p.indent(indent));
                str.println(indent + "</" + namespace + ":" + tag + ">");
        }else{
                str.println(indent + "<" + namespace + ":" + tag + nameAttr + "/>");
        }
}
}

public void FArraySubscripts.prettyPrint_XML(
                Printer p, PrintStream str, String indent, Object o){
String namespace = "exp";
String tag = this.xmlTag();

        str.println(indent + "<" + namespace + ":" + tag + ">");
        for (FSubscript s : this.getFSubscriptList())
                s.prettyPrint_XML(str,p.indent(indent+"\t"));
        str.println(indent + "</" + namespace + ":" + tag + ">");
}

public void FColonSubscript.prettyPrint_XML(
                Printer p, PrintStream str, String indent, Object o){
String namespace = "exp";
String tag = this.xmlTag();

        str.println(indent + "<" + namespace + ":" + tag + "/>");
}

public void FExpSubscript.prettyPrint_XML(
                Printer p, PrintStream str, String indent, Object o){
String namespace = "exp";
String tag = this.xmlTag();

        str.println(indent + "<" + namespace + ":" + tag +  ">");
        this.getFExp().prettyPrint_XML(str,p.indent(indent));
        str.println(indent + "</" + namespace + ":" + tag +  ">");
}
```

The code generation starts from the "FQName.prettyPrint_XML()" method, which calls the "prettyPrint_XML()" method for every qualified name part, i.e. for every "FQNamePart" node. Hence, the "FQNamePart" node calls the "prettyPrint_XML()" method for every subscript of the qualified name part, if any.

6.4 The ACADO Toolkit

The ACADO Toolkit [21] is a software environment and algorithm collection 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. ACADO Toolkit is implemented as self-contained C++ code and comes along with user-friendly Matlab interfaces. The object-oriented design allows for convenient coupling of existing optimization packages and for extending it with user-written optimization routines.

One of the basic problem classes which can be solved with ACADO toolkit are standard optimal control problems. These problems typically consist of a dynamic system with differential states and possibly also algebraic states, the objective can usually be written as a sum of a Lagrange and a Mayer term. Moreover, ACADO toolkit tackles several types of constraints, such as control and state bounds, terminal constraints, general nonlinear path constraints, periodic boundary conditions, etc

As an extension, ACADO toolkit offers systematic and advanced tools for solving general optimal control problems with multiple and conflicting objectives. Pareto frontiers (or trade-off surfaces) can automatically and efficiently be generated by several scalarization approaches, which convert the original multi-objective optimal control problem into a series of parametric single objective optimal control problems. The available scalarization approaches involve the classic convex Weighted Sum as well as recent techniques as Nor-

mal Boundary Intersection and Normalized Normal Boundary Intersection. Typical algorithmic features include smart re-initialization strategies for computational speed-ups and post-processing tools as Pareto filter algorithms.

An important class of optimal control problems, which obtains a special attention within the ACADO toolkit, are state and parameter estimation problems. This subclass of optimal control problems can theoretically also be transformed into a standard nonlinear optimal control problem. However, state and parameter estimation problems have often a certain structure, which can be used by the algorithms. In the ACADO Toolkit Gauss-Newton algorithms are implemented to deal with the least-squares terms, which typically occur within this class of optimization problems. Moreover, a-posteriori analysis tools are available such as a variance-covariance computation for the estimated states and parameters.

Finally, another highlight of the ACADO toolkit are its model based feedback control algorithms. The corresponding problems can be divided into two kinds of online dynamic optimization problems: the Model Predictive Control (MPC) problem of finding (approximately) optimal control actions to be fed back to the controlled process, and the Moving Horizon Estimation (MHE) problem of estimating the current process states using measurements of its outputs.

User manual and further documentation is available on ACADO Toolkit website [21].

6.5 Importing and reusing XML models in ACADO

The goal of the test case presented in this paragraph is to export a model from the JModelica.org platform and reuse it by the ACADO Toolkit. In particular an Optimica model is exported in XML format, in order to exploit also the feature of the extension presented in Section 4.4. The same results can be obtained by exporting a pure Modelica model and then manually

adding the information of the optimization problem by the ACADO Toolkit interface.

As a test an optimal control problem on the Van der Pol oscillator model has been taken into account. It requires to minimize the final time with respect of a constraint on the input signal. The optimization problem is described by the following Optimica code, available on the JModelica.org website [27].

```
package VDP_pack

 optimization VDP_Opt (objective = cost(finalTime),
                                                startTime = 0, finalTime = 20)

// Parameters
parameter Real p1 = 1;        // Parameter 1
parameter Real p2 = 1;        // Parameter 2
parameter Real p3 = 2;        // Parameter 3

// The states
Real x1(start=0);
Real x2(start=1);

// The control signal
input Real u;

Real cost(start=0);

equation
der(x1) = (1 - x2^2) * x1 - x2 + u;
der(x2) = p1 * x1;
der(cost) = exp(p3 * 1/*time*/) * (x1^2 + x2^2 + u^2);

constraint        u<=0.75;

end VDP_Opt;

end VDP_pack;
```

The raw Optimica model is then flattened into the following model.

```
optimization VDP_pack.VDP_Opt(objective = cost(finalTime),
        startTime = 0,finalTime = 20)
```

```
parameter Real p1 = 1 /* 1.0 */;
parameter Real p2 = 1 /* 1.0 */;
parameter Real p3 = 2 /* 2.0 */;
Real x1(start = 0,fixed = true);
Real x2(start = 1,fixed = true);
input Real u;
Real cost(start = 0,fixed = true);
Real der(x1);
Real der(x2);
Real der(cost);

initial equation
x1 = 0;
x2 = 1;
cost = 0;

equation
der(x1) = (1-(x2^2))*(x1)-(x2)+u;
der(x2) = ( p1 ) * ( x1 );
der(cost) = (exp(p3))*(x1^2+x2^2+u^2);

constraint  u <= 0.75;

end VDP_pack.VDP_Opt;
```

Finally an XML document representing the model is exported from the JModelica.org platform. The full XML document is given in Appendix B.

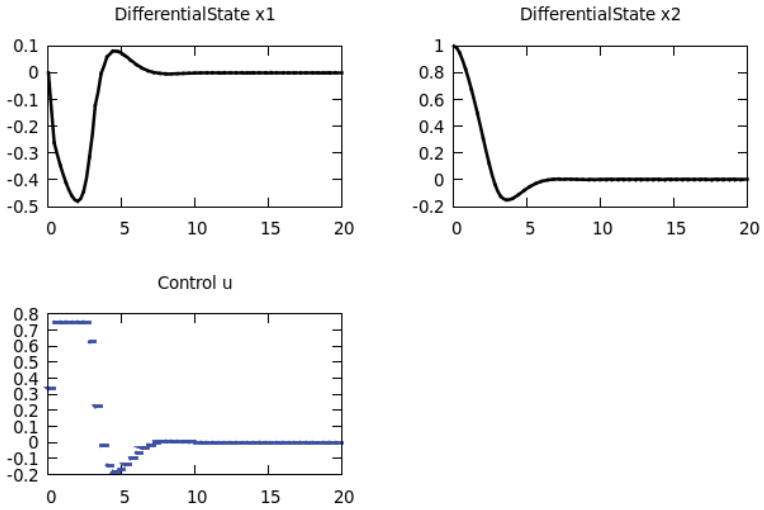
A simple XML parser has been implemented in the ACADO Toolkit platform. It directly transforms the XML code into the internal representation of the model used by ACADO Toolkit. Then the following code is used as input to import and solve the optimization problem.

```
OCPv2 get_fmi_OCP(const string& modelfile){
// Allocate a parser
FMIParser parser;

// Load the xml model
parser.loadFile(modelfile);

// Dump representation to screen
parser.dump();
```

Figure 6.2: Results returned by ACADO




```
// Create an optimal control problem
OCPv2 ocp;

// Get model from the xml file
parser.exportOCP(ocp);

// Print the ocp to screen
cout << ocp;

return ocp;
}
```

The optimal control problem has been parametrized as an non-linear problem using direct multiple shooting (with condensing) and solved by an SQP method (sequential quadratic programming) using qpOASES [16] as a QP solver. The results are given in Figure 6.2. The same results can be obtained by solving the problem by means of a collocation method available in JModelica.org.

Chapter 7

Conclusions and future perspectives

In this thesis, an XML representation of continuous time DAEs obtained from continuous-time Modelica models has been proposed. The test implementation on the JModelica 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 [11].

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 [9].

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

submodels to be connected should be designed.

Appendix A

Introduction to the XML Schema language

The intention of this appendix is to give a short summary on the key elements of the XML Schema language, that could be useful while reading the present report. A full and discursive presentation on the XML Schema language is given instead in [37].

XML Schema and XML document

An XML document is an XML file that represents an instance of a data structure. An XML Schema defines the rules that an XML document should hold to be valid for a certain application.Tools and libraries are available to verify that a certain XML document is valid respect to an XML schema.

Types and elements

Both types and elements define a portion of the schema. The main difference is that elements can be instantiated, while it is not possible to instantiate types. It is possible to assign a type to an element. Using more or less type definitions is a matter of design style and the main advantage is that

the maintenance of the schema is improved when many elements have a similar definition. As an example, in the expression module of the schema (see Section 4.2.4), BinaryOperation is the complex type representing binary operations, while addition, subtraction, etc, are elements of type BinaryOperation.

Namespaces

An XML namespace is a collection of XML elements and attributes identified by a standard URI name ("http://sampleaddress.com"). This collection is often referred to as an XML "vocabulary" and it works similarly as the packages in Java, allowing the elements and attributes to be reused and extended.

As an example, the schema module relative to expressions is defined in the following way:

```
<xs:schema
  xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
  elementFormDefault="qualified" attributeFormDefault="unqualified"
>
```

It is possible to distinguish two sections in this tag: the declaration of the vocabularies used in the schema and the definition of a target namespace. The "targetNamespace" attribute is the real definition of the namespace's URI. With xmlns:<prefix>="URI" it's possible to recall a vocabulary that will be used. naming it with a prefix. In this case two vocabularies are used: the one in the schema itself, referred with the prefix "exp" and the standard one (that it's used in every XML schema).

Note that the URI name is just a hint and there is no need that the schema is really located in the URI place (the address might even not exist...).

Wildcard elements

Wildcards are elements that can be replaced by every element defined into a specified namespace.

For example, the definition of the binary operation complex type (BinaryOperation) given in Section 4.2.4 is the following.

```
<xs:complexType name="BinaryOperation">
        <xs:annotation>
                <xs:documentation>
                        Binary operation complex type
                </xs:documentation>
        </xs:annotation>
        <xs:sequence>
                <xs:any namespace="##targetNamespace"/>
                <xs:any namespace="##targetNamespace"/>
        </xs:sequence>
</xs:complexType>
```

"BinaryOperation" complex type is composed by the sequence of two wildcard elements (the operands), that can be replaced by any element defined into the "targetNamespace", that is the "exp" namespace itself. If a new kind of operand (expression) will be needed, it will be necessary just to define it into the exp vocabulary and it will be automatically available as possible operand for every binary operations.

Composition of the schema: include, import and redefine

XML Schema language offers three possibilities to compose XML schemas:

- include : it is used to join schemas with the same target namespace;
- import: it is used to join schemas with different target namespace;

- redefine: it used to join schemas and the namespace of one schema is redefined as the target namespace of the including schema.

For example the expressions module is imported by the equations one by the following code:

```
<xs:import
  namespace="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
  schemaLocation="daeExpressions.xsd"
/>
```

The result is that the elements defined into the imported schema can be referred by their own namespace prefix (“exp”) and the equations by their own different one (“equ”).

Appendix B

Test case code

```
<fmiExtendedModelDescription
xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
xmlns:opt="https://svn.jmodelica.org/trunk/XML/daeOptimization.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
fmiVersion="1.0" modelName="VDP_pack.VDP_Opt"
modelIdentifier="VDP_pack_VDP_Opt"
guid="unsupported"
generationDateAndTime="2010-05-13T11:28:48"
variableNamingConvention="flat"
numberOfContinuousStates="3"
numberOfEventIndicators="0">

<ModelVariables>
        <ScalarVariable name="p1" valueReference="0" variability="parameter"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0" />
        </ScalarVariable>

        <ScalarVariable name="p2" valueReference="1" variability="parameter"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0"/>
        </ScalarVariable>

        <ScalarVariable name="p3" valueReference="2" variability="parameter"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0"/>
        </ScalarVariable>
```

```
        <ScalarVariable name="x1" valueReference="6" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0" />
        </ScalarVariable>

        <ScalarVariable name="x2" valueReference="7" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false"/>
        </ScalarVariable>

        <ScalarVariable name="u" valueReference="9" variability="continuous"
                causality="input" alias="noAlias">
                        <Real relativeQuantity="false" />
        </ScalarVariable>

        <ScalarVariable name="cost" valueReference="8" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0" />
        </ScalarVariable>

        <ScalarVariable name="der(x1)" valueReference="3" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0"/>
        </ScalarVariable>

        <ScalarVariable name="der(x2)" valueReference="4" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0"/>
        </ScalarVariable>

        <ScalarVariable name="der(cost)" valueReference="5" variability="continuous"
                causality="internal" alias="noAlias">
                        <Real relativeQuantity="false" start="0.0"/>
        </ScalarVariable>
</ModelVariables>

<equ:BindingEquations>
        <equ:BindingEquation>
                <equ:Parameter>
                        <exp:QualifiedNamePart name="p1"/>
                </equ:Parameter>
                <equ:BindingExp>
                        <exp:IntegerLiteral>1</exp:IntegerLiteral>
                </equ:BindingExp>
        </equ:BindingEquation>
```

```
        <equ:BindingEquation>
                <equ:Parameter>
                        <exp:QualifiedNamePart name="p2"/>
                </equ:Parameter>
                <equ:BindingExp>
                        <exp:IntegerLiteral>1</exp:IntegerLiteral>
                </equ:BindingExp>
        </equ:BindingEquation>

        <equ:BindingEquation>
                <equ:Parameter>
                        <exp:QualifiedNamePart name="p3"/>
                </equ:Parameter>
                <equ:BindingExp>
                        <exp:IntegerLiteral>2</exp:IntegerLiteral>
                </equ:BindingExp>
        </equ:BindingEquation>
</equ:BindingEquations>

<equ:DynamicEquations>
        <equ:Equation>
                <exp:Sub>
                        <exp:Der>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name="x1"/>
                                </exp:Identifier>
                        </exp:Der>

                        <exp:Add>
                         <exp:Sub>
                          <exp:Mul>
                                <exp:Sub>
                                 <exp:IntegerLiteral>1</exp:IntegerLiteral>
                                 <exp:Pow>
                                        <exp:Identifier>
                                                <exp:QualifiedNamePart name="x2"/>
                                        </exp:Identifier>
                                        <exp:IntegerLiteral>2</exp:IntegerLiteral>
                                 </exp:Pow>
                                </exp:Sub>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name="x1"/>
                                </exp:Identifier>
                          </exp:Mul>
                          <exp:Identifier>
```

```
                        <exp:QualifiedNamePart name="x2"/>
                    </exp:Identifier>
                </exp:Sub>
                <exp:Identifier>
                        <exp:QualifiedNamePart name="u"/>
                </exp:Identifier>
            </exp:Add>
        </exp:Sub>
</equ:Equation>

<equ:Equation>
        <exp:Sub>
                <exp:Der>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="x2"/>
                        </exp:Identifier>
                </exp:Der>
                <exp:Mul>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="p1"/>
                        </exp:Identifier>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="x1"/>
                        </exp:Identifier>
                </exp:Mul>
        </exp:Sub>
</equ:Equation>

<equ:Equation>
        <exp:Sub>
                <exp:Der>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="cost"/>
                        </exp:Identifier>
                </exp:Der>
                <exp:Mul>
                        <exp:Exp>
                                <exp:Identifier>
                                        <exp:QualifiedNamePart name="p3"/>
                                </exp:Identifier>
                        </exp:Exp>
                        <exp:Add>
                                <exp:Add>
                                        <exp:Pow>
                                                <exp:Identifier>
                                                        <exp:QualifiedNamePart
```

```
                                                        name="x1"/>
                                                </exp:Identifier>
                                                <exp:IntegerLiteral>2
                                                </exp:IntegerLiteral>
                                        </exp:Pow>
                                        <exp:Pow>
                                                <exp:Identifier>
                                                        <exp:QualifiedNamePart
                                                                name="x2"/>
                                                </exp:Identifier>
                                                <exp:IntegerLiteral>2
                                                </exp:IntegerLiteral>
                                        </exp:Pow>
                                </exp:Add>
                                <exp:Pow>
                                        <exp:Identifier>
                                                <exp:QualifiedNamePart name="u"/>
                                        </exp:Identifier>
                                        <exp:IntegerLiteral>2</exp:IntegerLiteral>
                                </exp:Pow>
                        </exp:Add>
                </exp:Mul>
        </exp:Sub>
    </equ:Equation>
</equ:DynamicEquations>

<equ:InitialEquations>
    <equ:Equation>
        <exp:Sub>
            <exp:Identifier>
                <exp:QualifiedNamePart name="x1">
                    <exp:ArraySubscripts>
                    </exp:ArraySubscripts>
                </exp:QualifiedNamePart>
            </exp:Identifier>
            <exp:IntegerLiteral>0</exp:IntegerLiteral>
        </exp:Sub>
    </equ:Equation>

    <equ:Equation>
        <exp:Sub>
            <exp:Identifier>
                <exp:QualifiedNamePart name="x2">
                    <exp:ArraySubscripts>
                    </exp:ArraySubscripts>
                </exp:QualifiedNamePart>
```

```
                        </exp:Identifier>
                        <exp:IntegerLiteral>1</exp:IntegerLiteral>
                </exp:Sub>
        </equ:Equation>

        <equ:Equation>
                <exp:Sub>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="cost">
                                        <exp:ArraySubscripts>
                                        </exp:ArraySubscripts>
                                </exp:QualifiedNamePart>
                        </exp:Identifier>
                        <exp:IntegerLiteral>0</exp:IntegerLiteral>
                </exp:Sub>
        </equ:Equation>

</equ:InitialEquations>

<opt:Optimization>
        <opt:ObjectiveFunction>
                <exp:TimedVariable>
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="cost"/>
                        </exp:Identifier>
                        <exp:Instant>20.0</exp:Instant>
                </exp:TimedVariable>
        </opt:ObjectiveFunction>
        <opt:IntervalStartTime>
                <opt:Value>0.0</opt:Value>
                <opt:Free>false</opt:Free>
                <opt:InitialGuess>0.0</opt:InitialGuess>
        </opt:IntervalStartTime>
        <opt:IntervalFinalTime>
                <opt:Value>20.0</opt:Value>
                <opt:Free>false</opt:Free>
                <opt:InitialGuess>1.0</opt:InitialGuess>
        </opt:IntervalFinalTime>

        <opt:TimePoints>
                <opt:Index>0</opt:Index>
                <opt:Value>20.0</opt:Value>
        </opt:TimePoints>

        <opt:Constraints>
                <opt:ConstraintLeq>
```

```
                        <exp:Identifier>
                                <exp:QualifiedNamePart name="u"/>
                        </exp:Identifier>
                        <exp:RealLiteral>0.75</exp:RealLiteral>
                </opt:ConstraintLeq>
        </opt:Constraints>
</opt:Optimization>

<fun:FunctionsList></fun:FunctionsList>

</fmiExtendedModelDescription>
```

Bibliography

[1] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, November 2007.

[2] Johan Åkesson. Optimica—an extension of Modelica supporting dynamic optimization. In *In 6th International Modelica Conference 2008*. Modelica Association, March 2008.

[3] 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.

[4] Johan Åkesson, Tove Bergdahl, Magnus Gafvert, and Hubertus Tummescheit. The JModelica.org Open Source Platform. In *7th International Modelica Conference 2009*. Modelica Association, 2009.

[5] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1-2):21–38, January 2010. doi:10.1016/j.scico.2009.07.003.

[6] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[7] U.M. Ascher and L.R. Petzold. *Computer methods for Ordinary Differential Equations and Differential Algebraic Equations*. SIAM, 1997.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] L.R. Petzold C.W. Gear. ODE methods for the solution of differential/algebraic systems. *SIAM journal on numerical analysis*, 21:716–728, 1984.

[15] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* Phd thesis, Department of Automatic Control, Lund University, 1978.

[16] 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.

[17] P. A. Fishwick. Using XML for simulation modeling. In *Winter simulation conference*, December, 8-11 2002.

[18] P. Friztson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* John Wiley & Sons, 2004.

[19] E. Hairer, Ch. Lubich, and M. Roche. *The numerical solution of differential-algebraic systems by Runge-Kutta methods*. Springer, 1989.

[20] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 1996.

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

[22] P. Kunkel and V. Mehrmann. *Differential-Algebraic Equations: Analysis and Numerical Solution*. European Mathematical Society, 2006.

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

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

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

[26] Modelon AB. Modelon AB Homepage. http://www.modelon.se.

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

[28] Constantinos C. Pantelides. The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

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

[30] Python Software Foundation. Python Homepage. http://www.python.org.

[31] 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.

[32] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[33] T. Ekman, G. Hedin. JASTAdd Homepage. http://www.jastadd.org.

[34] The Modelica Association. Modelica Association Home Page. http://www.modelica.org.

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

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

[37] Priscilla Walmsley. *Definitive XML Schema*. Pearson Education (US), 2001.

[38] World Wide Web Consortium (W3C) . Extensible Markup Language (XML). http://www.w3.org/XML/.

[39] World Wide Web Consortium (W3C) . XML Schema (XSchema). http://www.w3.org/XML/Schema.