GÖTEBORGS UNIVERSITET

XML export and import of Modelica models

Master of Science thesis in Computer Science

NIKLAS LANDIN

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden 2014
Master of Science thesis 2014:09

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

XML export and import of Modelica models

NIKLAS LANDIN

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2014

XML export and import of Modelica models

Examiner: Bengt Nordström

Master of Science thesis 2014:09
University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2014

Abstract

Modelica is a language primarily used for simulation of physical systems but other use cases such as optimization and control design are increasing. New use cases introduce a different set of algorithms to work with the models, which means that current Modelica tools can not implement all these algorithms. Instead it is important to be able to exchange models between tools. To solve this issue a standardized XML format called Modelica XML is under development.

In this thesis it is investigated how suitable Modelica XML is for transfer of Modelica models between different tools. As a part of the evaluation of the Modelica XML format a comparison with a previous XML format used for model exchange is performed. An implementation of import and export for the format within the open source Modelica compiler JModelica.org is performed to test the functionality and performance.

The results from the implementation show significantly smaller XML for Modelica XML compared to the old format. Furthermore it also demonstrated that the required functionality of the format for transfer of models is present. The comparison indicates that Modelica XML is a more expressive and versatile format than the older format. A conclusion that Modelica XML is a suitable format for representation of Modelica models and model exchange is presented.

Keywords: XML, Modelica, JastAdd, Modelica XML

Acknowledgements

I would like to thank my advisor Toivo Henningsson at Modelon for his advice and support during the thesis. Furthermore I would like to thank Modelon for giving me the opportunity to carry out this thesis and finally I would like to thank my advisor at the university, Bengt Nordström.

Contents

1 Introduction

Modelica is a language for modeling of physical processes using differential-algebraic equations. Simulation of systems is the major usage, but Modelica models are increasingly used in other applications such as optimization and control design[1][2]. New applications introduce a whole different set of algorithms that are needed to work with the models. A growing number of algorithms means that Modelica tools will not be able to implement all these algorithms, so it becomes important to be able to exchange models on symbolic form among different tools. To satisfy this need a standardized XML format is under development within the European research project MODRIO, the standardization process is lead by Modelon.

In this thesis the new XML format called Modelica XML (MXML) will be evaluated and compared with existing formats for model transfer. *JModelica.org* is a Modelica based open source platform for simulation and optimization of complex dynamic systems[3]. As a part of the evaluation the JModelica.org compiler is extended with XML generation for the MXML format. Since the MXML format is intended to be used for transfer of models the thesis will also consider import of the generated XML into a runtime toolchain for optimization called *CasADiInterface*.

1.1 Purpose

The major motivation for this thesis is to extend the possibilities to transfer models among different Modelica related tools. A model in Modelica refers to a representation of a problem with Modelica code. The use of Modelica is growing and as new areas of applications are found, the current mechanisms for transferring models among different domains are not sufficient. *JModelica.org* supports several different export formats but these formats was designed for export to specific tools and a more generic alternative is needed. A standardized format for representation of Modelica models would provide such a generic solution.

The need for this can be seen in the *JModelica.org* system, where the current import into *CasADiInterface* is based on extracting a model directly from an instance of the compiler. This approach may work but it is inconvenient and inefficient since it requires the *JModelica.org* compiler to run each time a model is transferred. With the MXML format in place it would be much easier to generate an XML representation of the model and then make the import from this XML document.

1.2 Goals

The goal of this thesis is to investigate whether the MXML format is a good format for representing and exchanging Modelica models among tools and how it compares to the previous formats that where used for similar tasks. The thesis also aims to show that MXML can provide the necessary functionality as well as improvements over other formats that are used for the same purpose.

As a part of this goal the thesis aims to extend the open source Modelica compiler *JModelica.org* with support for the MXML format. This includes implementation of a compiler backend for XML generation as well as implementation of an import module into the *CasADiInterface* that is under current development. Currently *JModelica.org* supports several techniques for

transferring subsets of models among different tools, but it lacks a way of transferring complete models. The implementation is an essential part in showing that the MXML format is suitable for transfer of models and that it can be used within a complex system.

1.3 Method

Initially a literature review was conducted with focus on previous work that was related to the *JModelica.org* system and XML formats for representation of Modelica models. The XML schema description of the MXML format was studied to understand how the format was structured and how it represented specific parts of a Modelica model. This study coincided with reading about the Modelica language since it was an essential part to understand the MXML format. To prepare for the implementation the code structure and techniques used within *JModelica.org* was studied and a basic structure for the implementation was proposed.

The implementation was conducted in an iterative manner where features was added and then tested. Most of the export part was implemented before the work on the import started. The import was implemented together with the final parts of the export since some design decisions interfered with both parts.

Towards the end a comparison of the MXML and FMUX formats where conducted. This included reading the reports about the FMUX format and analyze both the formats. The implementation was tested to get some data about how the MXML implementation compared to the other alternatives with regards to performance and functionality. Some tests that compared the performance of the MXML and FMUX where also conducted. The final part of the work was to combine the results of the comparison and the implementation to a final evaluation of the MXML format.

1.4 Scope

The comparison of formats in this thesis is limited to preexisting XML formats that are used for model transfer. It would be interesting to develop a custom format that is designed with the purpose of representing Modelica models and compare it to the MXML representation. However due to the time required to make such a format it is out of scope for this thesis.

Modelica is a complex language that supports a large number of features and to develop XML export and import for all these features are outside the range of this project. This thesis will only consider export and import of so called flat models, these are models that are purely symbolic which means that they do not contain any hierarchical structures.

1.5 Outline

The thesis is divided into six chapters. Chapter 1 introduces the motivation behind the thesis and the goals. In chapter 2, a background of some related work is given followed by descriptions of the tools and techniques used in the implementation. The third chapter provides a comparison between the MXML format and the currently used exchange format, FMUX. Chapter 4 describes the implementation, it starts by giving an overview of where the implemented parts are located in the JModelica.org system. This is followed by a description

of the implementation of the XML code generator and finally a description of the import into the *CasADiInterface*. In Chapter 5 several tests of the implementation are presented together with the results of the tests. Chapter 6 gives an evaluation of the implementation and the MXML format followed by a final conclusion and some remarks on possible future work.

2 Background

In this chapter some background on previous work is provided as well as an introduction to the technologies and tools that form the basis of this thesis. First the MODRIO project is introduced followed by some background on why XML was the chosen format for MXML. In the next section the Modelica language is described. Then the open source Modelica compiler *JModelica.org* will be introduced together with the underlying meta-compilation tool JastAdd. Finally the algebra system CasADi that is used within the *CasADiInterface* is explained in general and the symbolic framework used within the import is described in particular.

2.1 MODRIO and related work

MODRIO is a research project that aims to extend state of the art modeling and simulation environments to increase dependability and performance throughout their lifecycles. The project involves 38 partners from six different European countries and is a part of the ITEA programme. Development and standardization of MXML is a part of a MODRIO financed project lead by Modelon[4][5].

Within the predecessor of MODRIO, OPENPROD, several interesting results where reached that relate to this thesis. One example is the extension of the *OpenModelica* compiler with generation of XML outlined in the thesis by Shitahun [6]. This closely resembles the generation that this thesis aims to extend the JModelica.org compiler with but for another format than the MXML format. This work was expanded upon and the possibility to transfer optimization problems using XML and solve them with CasADi from the OpenModelica system was shown in [7].

2.2 Data serialization formats

There exist several different data serialization formats that could be used for representing Modelica models. Some of the more common are JavaScript Object Notation (JSON), YAML and XML. Each of these formats have different strengths and disadvantages and to get a better overview over each of them the formats are briefly described below. A motivation to why XML was chosen is then given in section 2.2.4.

2.2.1 JSON

JSON is a data serialization format where the data are represented by two different structures. The first structure is a collection of name-value pairs, which can be represented either with objects or records in most programming languages. The second structure is an ordered list of values, which can be represented by an array or list. Since the data representation in JSON closely resembles constructs used in most programming language the data can trivially be represented within most programming languages.

JSON is a lightweight format that lacks any markup, which makes it smaller than XML and other markup based formats. There exist several tools for a wide variety of languages that provide functionality to parse JSON into language specific objects or constructs. Compared to

XML that can be used for several different tasks, JSON is designed with one purpose and that is as a data interchange format.

2.2.2 YAML

YAML is a data serialization format that is designed to be human readable. YAML has a structure that resembles JSON and in fact JSON syntax is a subset to YAML, which means that some YAML parsers can parse JSON documents. Even though the formats are similar YAML supports a number of features that JSON does not, for example user-defined types and comments. In YAML whitespace is used to denote the structure of a document. This means that no brackets or braces are used, which is an intentional design decision to increase readability. YAML also supports references to other items, which makes it possible to represent relational data[8].

2.2.3 XML

XML is a markup language that is commonly used as a data-interchange format. An XML document is divided into markup and content, to differentiate between markup and content some simple syntactic rules are used. Everything that is contained between a < and a > is considered markup. This entity is referred to as a tag. There are three different types of tags, start tags, end tags and empty tags. An element is the content between a start tag and an end tag. The markup parts of the XML add a substantial overhead that is avoided in both JSON and YAML since they are specifically designed for data serialization and not as markup languages.

XML supports validation of documents by providing a reference in the document to a specification. To specify the allowed structure of an XML document a schema language is used. The two most common are *Document Type Definition* (DTD) and *XML Schema Definition* (XSD). DTD is the oldest schema language and XML has syntactic support for embedding DTD:s. XSD is a newer schema language that supports more features than the older DTD. More detailed constraints on the structure of the XML document are supported as well as a more complex type system[9].

2.2.4 Motivation

There are several factors that contributed to XML being the format that MXML was based on. One primary reason was that XML is a well-established format and that a number of XML formats for different subsets of Modelica already exist[10][11]. Another reason was that the format needed to be extensible and with an XML schema it is easy to add new features. Furthermore XSD is a mature technique for specification and it has a proven track record[12].

The main disadvantage with XML over the other formats are that it is verbose and the size of the documents will be larger than if the other formats where used. Case studies show that transfer of JSON objects is faster than transfer of XML objects[13]. JSON:s speed advantage in the study is likely a consequence of more lightweight objects than the XML. It is likely that YAML objects would be of smaller size than XML as well, albeit not as lightweight as the JSON objects.

Another approach than using any of the common formats could be to design a custom format. The main problem with a custom format is that it would require more work to add support for

the format in new tools. For XML there exist many tools for parsing and manipulation but with a custom format each implementation would need to handle these problems by themselves. Since the developed format is intended to work on a lot of platforms and in many systems it would require too much work. However if the format was intended to be used only within the same tool a custom format could be an appropriate solution.

In summary, it is clear that the decision to chose XML was based on the premise that a well-established and extensible format was needed. Size and performance were not considered as the primary factors when the format was chosen. From a pure performance aspect the best decision would probably be to design a custom format, however as mentioned earlier this would introduce other problems. The use of XML as the format provides the possibility to easily add metadata about the model, which is something that neither JSON nor YAML is designed to support.

2.3 Modelica

Modelica is an object-oriented modeling language used for modeling of complex systems using hybrid differential algebraic equations. It is provided for free and is developed by the non-profit organization Modelica Association. Modelica has been used in industrial projects since 2000 and several new versions of the language have been released since the first version in 1997. Modelica was initially described in the PhD thesis of Hilding Elmqvist and he is seen as the architect of the language[14].

Modelica is not a programming language but rather a modeling language, although it is syntactically similar to traditional programming languages it has little in common with them. Modelica models are not compiled in the traditional sense, instead they are converted into objects that are handled by a simulation engine. A Modelica model usually consists of several mathematical equations that represent some kind of physical system. Within Modelica an equation is used to describe equality between two mathematical expressions. When working with Modelica it is important to know that the order of the equations in the source file may not be the same as their order of execution. The simulation engine may symbolically manipulate the equations to determine their order of execution. Modelica requires that for each variable in the model there must exist one equation, this is to guarantee that the equation system is solvable.

The basic building block of the Modelica language is a model, which is similar to a class in other object-oriented languages. A Modelica model consists of a set of variable declarations followed by an equation section, an algorithm section and an arbitrary number of functions. A simple example of how a model might look is given in listing 2.1. Modelica has support for object oriented features such as classes, generic types and inheritance. These are key components of the language that makes it able to construct complex systems by connecting different submodules. The features offered by Modelica make it easy for a modeler to reuse existing code and it promotes the idea of modular code.

Equation	Description
=	Equality equation that expresses equality between two expressions.
If	Conditional equality equation
For	For equations is used to define equality among a number of variables within a loop
Connect	Connect equations are used to introduce connection among objects
When	Construct used to define behaviour of events

Table 2.1: The available equation types in the equation section

Listing 2.1: Example of a simple Modelica model, notice how the derivative funtion is used on the left-hand side of the second equality equation.

```
1 model Example
2   Real x;
3   Real y;
4 equation
5   x = 2+4;
6   der(y) = sin(x);
7 end Example;
```

Modelica supports four built-in types: `Real`, `Integer`, `Boolean` and `String` and has constructs for creating new types. A Modelica variable consists of a name and a type. Variables have a variability attribute that defines how the value of the variable may vary. For example a discrete variable may only change at discrete events whereas a constant variable cannot change its value at all during execution. It is possible to set attributes on variables when they are declared, some of the more common are `start` that sets the start value of the variable in the simulation and `min/max`, which set a minimum/maximum value that a variable can have. Variables are always declared at the beginning of the model except for function variables that are declared within the function.

The behavior of a model is defined within the equation and algorithm sections, as mentioned earlier equations are used to define equality between two physical variables. To capture this in a traditional programming language is a hard and tedious task. In Modelica this is easy due to the possibility to declaratively state the equations on their natural form and then let the simulation engine handle the difficulties. In table 2.1 the available equation types are listed. The most common is the `=`, which is used to express equality between two expressions.

The reason for having an algorithm section is that not everything is convenient to express using equations. In algorithm sections traditional programming constructs such as assignments, conditional statements, loops, break and return statements are supported. The assignment operator is defined as `:=` and it works as in most other programming languages. A function consists of a number of variables together with an algorithm section that contains the functionality of the function. Functions can have an arbitrary number of inputs and outputs, which are defined by variables with a *causality* attribute set to either *input* or *output*. Local variables within a function is also supported and are created by omitting the casuality attribute for the variable.

For a more detailed description of the Modelica language and what it supports, see the official Modelica specification [15].

2.3.1 Optimica

Optimica is an extension of the Modelica language that enables the specification of dynamic optimization problems based on Modelica models[16]. To represent optimization problems Optimica introduces a new class called *optimization*, which allow several new constructs that are unique for Optimica and only valid within the class. The *optimization* class can contain traditional Modelica constructs such as component and variable declarations, classes, equations and functions.

Optimica extends the Real type with two new attributes, *free* and *initialGuess*. The free attribute determines if a variable is free during the optimization and the initialGuess attribute provides an initial guess for a variable's value. To represent global properties of the optimization problem Optimica adds four class attributes. They are *objective* that defines the cost function, *startTime* that give the start time of the optimization, *finalTime* that gives the end time of the optimization and finally *static*, which states whether the problem is static or dynamic.

To represent constraints on variables a constraint section is introduced. Optimica supports two types of contraints, path constraints and point constraints. Path constraints are constraints that must be fulfilled during the whole simulation whereas points constraints only need to be fulfilled at a particular point in time. A constraint consists of an expression with an equal, less than or greater than clause.

2.4 JastAdd

JastAdd is a compilation tool designed to support modular development of compilers and other tools like source code analyzers. The basic structure of a JastAdd project is a class hierarchy that represents an abstract syntax tree. This class hierarchy is generated from a grammar specification of a language. Abstract syntax trees are commonly used in compiler development because they give a convenient representation of a program to work with[17].

JastAdd provides a way to extend a node in an abstract syntax tree with some behavior using aspects. An aspect is a module that uses inter-type declarations to add methods to nodes in the tree. This gives a logical and convenient way of adding a specific behavior to several classes instead of adding it separately to each of them. To propagate information among nodes in the abstract syntax tree JastAdd has two constructs, inherited and synthesized attributes. Synthesized attributes compute information in some node and that information can then be propagated upwards in the tree whereas inherited attributes pass information downwards in the tree. Furthermore, JastAdd allows attributes to have references to other nodes in the abstract syntax tree. This is a powerful feature that allows different places of the tree to be directly connected in a graph structure instead of the traditional tree structure[18].

2.5 JModelica.org

JModelica.org is an open source platform for simulation and optimization of complex dynamic systems using Modelica. It contains a compiler for the Modelica language, integration with algorithms for solving large dynamic optimization problems and a Python environment for easy integration and manipulation of models. *JModelica.org* is a result of research at the department of Automatic Control at Lund University and is now maintained by Modelon in collaboration

with Academia[19].

2.5.1 Flat model

Within the compiler several different representations are used, the final representation that is used for code generation is called the flat model. Compared to a regular Modelica model a flat representation lacks any hierarchical structures. For example consider an array `a` with two elements. In the flat model this array would be represented by two variables with the following names: `a[1]` and `a[2]`. Note that `a[1]` is an actual variable and not an element in the array. The flat model is represented by the `FClass` class in the abstract syntax tree, which contains all variables, equations and functions of a model.

2.5.2 Compiler

The compiler in *JModelica.org* is built on top of the meta-compilation tool JastAdd, which is further described in section 2.4. The compilation of a Modelica model is divided into three different phases. In the first phase the model is parsed and an abstract syntax tree is constructed. After the initial abstract syntax tree is constructed a flattening of the model occurs and a flat model is obtained. The flat model is used in the second phase where the compiler manipulates it by sorting the equations, index reduction and tearing. This manipulation is performed to gain efficiency in the third and final phase of the compilation. In the final phase the compiler generates simulation code from the flat model in either C or XML format. The generated code can be used for simulation or optimization of the model with the internal support that *JModelica.org* has. It may also be used for exporting models to other tools to work with them there. The compiler supports code generation for three different formats; JMI, FMI and FMUX. They are briefly described in the following sections.

The compiler is structured into different parts. There is a frontend part that handles parsing of Modelica models and construction of the abstract syntax tree. Type checking is also a part of the frontend before the abstract syntax tree is retrieved to the backend. The backend part is divided into several different modules for code generation. They are glued together with the frontend by the main class `ModelicaCompiler`, which is responsible for providing the functionality towards the end-user.

2.5.3 FMI

The Functional Mockup Interface is a standard developed to support exchange of Modelica models. *JModelica.org* supports both import and export of FMI code units, so called Functional Mockup Units (FMU). An FMU consists of an XML file that contains the variable declarations of a model and C files that contain simulation code. FMI provides a standardized interface for gaining access to the information in FMU:s, which allows other tools to be able to simulate the models[20].

The XML format that FMI uses to represent variables is also used in the FMUX format that is described in section 3.1. Figure 2.1 describes the top-level structure of the format. The starting element is `fmiModelDescription`, which has several attributes that specify general information about the model such as `modelName`, `modelIdentifier` and `author`. The `fmiModelDescription` has several optional children, `ModelVariables` is the most relevant since this is used to represent the variables in a model. `UnitDefinition` defines a set of units for

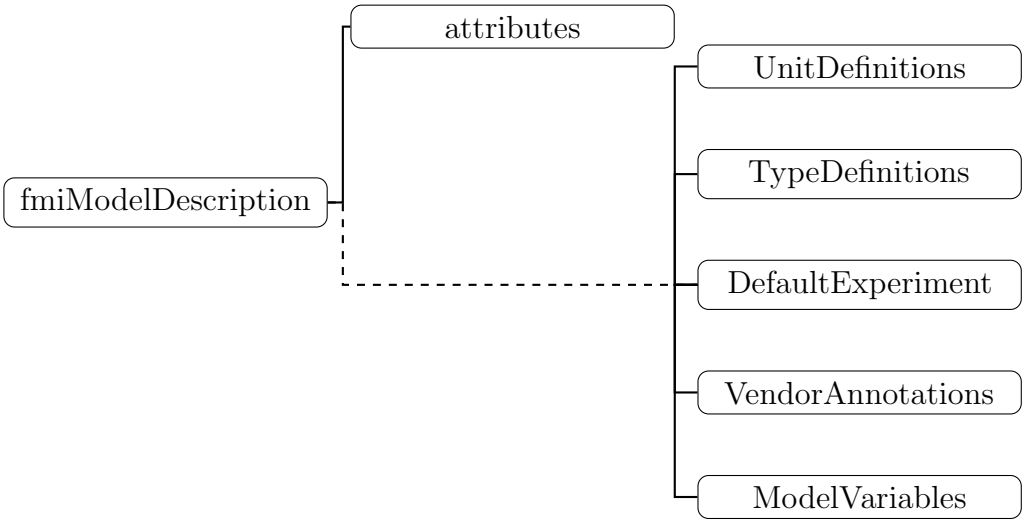

Figure 2.1: Structure of the XML used in an FMU

Attribute	Description
name	The name of the variable
valueReference	Value used to identify variable in a function call
description	Short description of the variable (optional)
variability	The variability of the variable (optional)
causality	The causality of the variable (optional)
alias	Alias name of the variable (optional)

Table 2.2: The attributes of the `ScalarVariable` *element in the FMI with brief descriptions*

converting display units into the units used by the equations in the model. `TypeDefinitions` defines a set of type definitions that are used by the variables of the model. `DefaultExperiment` contains simulation specific data about the model. `VendorAnnotations` can be used by different vendors to provide specific information that is required for their tools.

`ModelVariables` consists of several `ScalarVariable` elements. The `ScalarVariable` element is used to represent one model variable. In table 2.2 the available attributes to the `ScalarVariable` element is given. The type of the variable is represented by a child element. A variable may be of the following types: *Real*, *Integer*, *Boolean*, *String* or *Enumeration*.

2.5.4 JMI

JMI is short for JModelica.org Model Interface, which is an internal format used within *JModelica.org* for simulation and optimization. In a similar fashion to FMI, JMI defines a JModelica.org Model Unit (JMU). A JMU consists of the DAE of a model and the models optimization information. To gain access to the information stored within the JMUs a number of methods are provided by JMI.

2.5.5 FMUX

FMUX is a symbolic format for exchange of continuous time differential algebraic equation systems. This format is the result of a former master thesis work at Modelon. The format

itself is first outlined in a paper by Casella, Donida and Åkesson[21] and then the construction of the format is completed as a master thesis by Parrotto [11]. FMUX is built as an extension of the XML format used in the FMI standard. The FMUX format is described more in detail in section 3.1

2.6 CasADi

CasADi is a minimalistic open source computer algebra system for numerical optimization using automatic differentiation [22]. CasADi is implemented in C++ but has front ends in Python and Octave. It supports several different flavors of automatic differentiation and is integrated with state of the art solvers such as IPOPT, Sundials, and KNITRO. In the thesis CasADi is used within the *CasADiInterface* for optimization of models.

CasADi has two different symbolic classes for representing expressions, SX and MX. SX can only be used for expressions that are scalar whereas MX allows sparse matrix-valued output and input functions[23]. The main difference between SX and MX is that MX is more expressive while SX is faster Since MX is the one used in the optimization toolchain it is described more in detail below.

2.6.1 Use of MX symbolic

CasADi provides a class `MX`, which makes it possible to build up MX expressions. The `MX` class provides a wide variety of methods that can be used when working with MX expressions. For example all common mathematical operators as well as methods for checking properties can be reached through the `MX` class. In listing 2.2 a simple example of an addition between two MX expressions is given. The `MX` class has constructors for scalar constants and symbolic matrixes. A symbolic matrix is constructed by giving a string with the name in the constructor instead of a scalar value.

Listing 2.2: Example of how an MX expression is constructed in C++

```
1 MX a = MX("a"); // a new variable
2 MX b = MX(5); // a new constant
3 MX c = a+b;
```

MX can represent function calls, but to represent functions something more elaborate is required. To solve this the `MXFunction` class is used which gives a way of representing functions with MX expressions. To construct an `MXFunction`, two vectors with MX expressions are passed as parameters to the constructor. The first vector consists of the input expressions and the second vector contains the output expressions. The `MXFunction` class provides methods for working with the function. The most important methods are the `init` method, which initialize the object, the `evaluate` method that performs a numerical evaluation of the function and the `call` method that is used for symbolically calling the function.

3 Comparison of FMUX and MXML

In this chapter the MXML format is compared with the FMUX format. Both formats are used to describe Modelica models but they have significant differences. MXML supports all types of models whereas FMUX only support models that are continuous in time. This means that discrete and conditional models are not supported by the FMUX format. In MXML all data are represented by attributes whereas in FMUX elements are used to represent some data. FMUX only supports a small subset of the available equation constructs. In MXML all types of equations are supported.

The chapter starts with a description of how the FMUX format is structured and continues with a more detailed description of the individual parts of the format. The following section describes the MXML format in a similar fashion. After both formats have been introduced a comparison of their properties and layouts are presented in the final section.

3.1 FMUX format

FMUX is based on the structure of the XML format used in FMI. Since the FMI format only supports a representation of scalar variables it is extended with support for qualified names, expressions, equations, functions and algorithms. The FMI schema only supports scalar variables whereas FMUX must support array and record variables as well. This is handled by introducing `QualifiedName`, which contains the member names and an optional number of array subscripts. Each of the other extensions are represented by a namespace, expressions by the `exp` namespace, equations by the `equ` namespace, functions by the `fun` namespace and algorithms by the `alg` namespace. Optimization problems are also represented by a separate namespace `opt`.

Expressions

The expression namespace is structured into six different groups of expressions, which are outlined in table 3.1. Note that apart from these groups there are a few special cases that are defined separately such as the array constructor and the range function. As seen from the table two different structures are used for representing mathematical operators, one for unary operators and one for binary operators. For built-in functions `Builtin1or2Funct` and `Builtin2Funct` are used; it should be noted that the latter is only used for mathematical functions whereas the other represents all other types of built-in functions. The schema distinguishes between mathematical operators and functions; `+` is considered an operator and `sin` is considered a function. `FunctionCall` represents function calls to user defined functions, child elements are the name of the function as well as the arguments.

Functions

The function namespace is structured into a root element that defines the starting point of the function and then the elements are defined in order. In table 3.2 the child elements are listed together with a short description. Most of the elements are self explanatory but some of them need further clarification. Input and output to a function is represented by two sets of variables, `InputVariable` and `OutputVariable`. `ProtectedVariable` represents the variables that are

Expression	Description
Exp	Base expressions such as literals
UnaryOperation	Mathematical or logical built-in unary operators
BinaryOperation	Mathematical built-in binary operators
Builtin2Funct	Built-in mathematical function with two arguments
Builtin1or2Funct	Built-in function with one or two arguments
FunctionCall	User-defined function calls

Table 3.1: Table over the available expression groups in the FMUX

Element	Description
Name	The name of the function
OutputVariable	Variables that contain the output of function
InputVariable	Variables with input to function
ProtectedVariable	Variables that are internal to the function
Algorithm	Algorithm section of the function
InverseFunction	The inverse of the function
DerivativeFunction	Derivative of the function

Table 3.2: Children element to the function root element

internal for the function. The two optional parts `InverseFunction` and `DerivativeFunction` declare an inverse or derivative to the function.

The algorithm section of a function contains all operations that occur in the function. Since there are many constructs that are allowed in the algorithm section it has a separate namespace and its content is further described below.

Equations

The equation sections of a Modelica model are divided into three different types in the FMUX schema. The first one being the `BindingEquations`, which represents binding equations of parameters in the model. The second is `DynamicEquations` which handles regular equations as well as equations with function calls. `InitialEquation` is the third type and is used to represent the initial equations of a model.

The `BindingEquation` type contains two children, which represent the left-hand and right-hand side of the equation. The left-hand side is represented by the `Parameter` element and the right-hand side is represented by the `BindingExp` element. Both the `Parameter` and `BindingExp` elements consists of expressions.

`DynamicEquations` consists of several `Equation` and `FunctionCallEquation` elements. The `Equation` element is of `AbstractEquation` type, which is used for representing equations in residual form. An equation in residual form has the structure $lhs - rhs = 0$. In FMUX this is represented by a substitution of the left-hand side with the right-hand side of the equation. The `FunctionCallEquation` type does not represent a function call in residual form. Instead the function call is handled by having an `OutputArgument` and a `FunctionCall`, where `OutputArgument` represents the outputs of the function call and `FunctionCall` represents the call with a name element and an argument element.

`InitialEquations` defines an `Equation` element, which is of the type `AbstractEquation`.

Element	Description
Assign	Assign statement, e.g., := in Modelica
Break	Break statement, only used in loops
Return	Return from function
If	If statement, can contain elseif and else
While	While loop
For	For loop
FunctionCallStatement	Function calls within algorithm section
Assertion	The assertion call

Table 3.3: FMUX structure of algorithm namespace

In Modelica it is allowed to have function calls in initial equations. This is not allowed in a FMUX and therefore the `InitialEquations` only consists of a number of `AbstractEquations`.

Algorithms

The `alg` namespace defines the statements that are valid in an algorithm section. Table 3.3 lists the element in the algorithm section. The assign statement that represents Modelica's `:=` construct has two children, an identifier as the left-hand side and an expression as the right-hand side. Break and return statements are represented with elements that have no children or attributes.

To represent conditional statements, the complex type `ConditionalStatement` is used. It defines a condition element and an arbitrary number of statement elements. This is sufficient for representing `while` and `if` statements. An `elseif` branch of an `if` statement consists of a `ConditionalStatement` just as the first `if`. The `else` branch consists of an arbitrary number of statements. A `for` statement is represented by several statements as well as an `Index` element. The `Index` element consists of a `QualifiedName` of the variable and an expression defining the set of possible values that the variable could iterate over.

Optimica

The constructs that are specific for an optimization problem are contained within the `opt` namespace. The `opt` namespace consists of an `Optimization` element that is the root for the optimization problem. There are a number of child elements to represent the different constructs that exist in an optimization problem. Constraints are represented by the `opt:Constraints` element, which in turn has the child elements `opt:ConstraintEq`, `opt:ConstraintGeq` and `opt:ConstraintLeq`. Each of these constraint types contain two expressions that define the different sides of the constraint. To represent the start time and final time of the optimization problem the `opt:IntervalStartTime` and `opt:IntervalFinalTime` elements are used. The objective functions are represented by the two elements `opt:ObjectiveFunction` and `opt:IntegrandObjectiveFunction`, which consist of one expression that defines the objective.

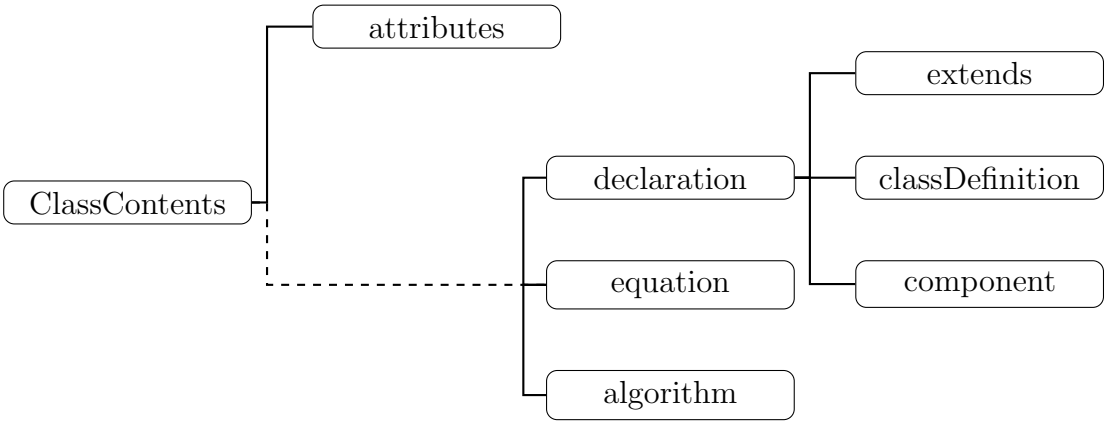

Figure 3.1: Root structure of the Modelica XML schema

3.2 Modelica XML

The MXML format that is under development aims to provide a standardized way of representing Modelica models in XML. MXML is designed to support all major constructs of Modelica and can therefore be used to model any Modelica model, which was not possible by the FMUX format.

The format is structured into three XML schemas, each describing the structure of one part of the XML. The first schema defines the general structure of the document. Figure 3.1 shows the basic structure of how the schema represents a Modelica model. `ClassContents` is used to represent the model, but it can also be used to represent other entities such as functions. The dashed line shows that the underlying elements are optional. One attribute is required, the `kind` attribute that is used to define the type of `ClassContents`. For a regular model the `kind` would be *model*. The content of the model would then be defined by three optional elements. First the variable declarations and new types are defined in the `Declaration` group. This is followed by an equation section and the final part of the content is an algorithm section. Both of these are defined in a separate schema and are explained later on in the section.

To represent declarations three different constructs are used within the `Declaration` group. First the `extends` element that is used to represent extends clauses. Secondly the `classDefinition`, which is used to represent declarations of enumerations, user-defined types and functions used in the model. Finally the `component` element that represents variable declarations in the model.

Equations and algorithms

The second schema defines the structure of equation and algorithm sections. Since the structure is similar and several constructs are common for both sections they are defined in the same schema. An equation can be of one of the seven different types that are listed in figure 3.2. The equal element is used to describe equality between two expressions, corresponding to the = operator in Modelica. Function calls that can occur in equations are represented by two different elements, the operator and call elements. The reason for having two different elements for representing function calls are to distinguish between purely mathematical functions and others. For example `der()` is an operator since $\frac{dx}{dt}$ is not a mathematical function of at a given time point. There are two conditional clauses, if and when. The if clause is represented as

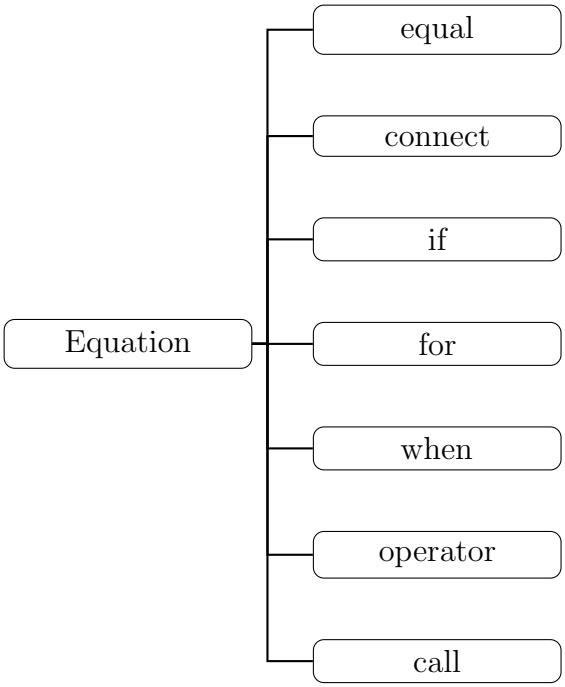

Figure 3.2: Elements that can be contained in the equation

an element with several conditional branches and then branches followed by one optional else branch. The when clause has the same structure but does not contain an else branch.

The definition of algorithms is the same as the equation definition with the some minor differences. Algorithm sections do not support the use of the `equal` element, instead the `assign` element is added where an expression is assigned to a variable. Three other elements are also added `while`, `break` and `return` constructs that are allowed in algorithm sections but not in equation sections. Break and return are represented by a stand-alone element. The while element is similar to a regular if, with the difference that there is no else clause and the name of the root element is changed to `while`.

Expressions

The third schema defines expressions, in table 3.4 the groups and elements of the expression schema is shown. The `Literal` group represents the literals in Modelica. It consists of five elements; `real`, `integer`, `true`, `false` and `string`. Except for the boolean values each one of them has an attribute *value* that gives the literal value. The `if` clause supported in the expression schema is exactly the same as the `if` clauses supported in the equation schema with the difference that the then and else clauses are expressions instead of equations.

The `call` construct is used to represent built-in calls as well as user-defined function calls. The `call` element has an optional attribute `builtin` that is used to represent a built-in call. If it is a call to a user-defined function an optional child element `function` is used instead of the `builtin` attribute. The argument to the functions are provided as children elements to the `call` element. The final group `LValue` refers to variables used in expressions. For example in a declaration of the following type: $x = 4$, x would belong to the `LValue` group whereas 4 would be a literal. The `LValue` group also contains a tuple element that is used to represent a tuple of variables.

Element	Description
Literal	Group that defines a set of literal
if	If expressions
call	Function calls
operator	Built-in operators e.g., `der()`
LValue	Group that defines variable references

Table 3.4: Expression group/elements in ModelicaXML

MXML distinguishes between built-in functions and has a special element `operator` that is used for functions that does not only depend on their input. For example consider the `der(x)` function, which takes a real value `x` as parameter. For a regular function call the return value would only depend on the input `x` but since `der(x)` is the time derivative of `x` it is in general independent of the current value of `x`.

Functions

Functions are not represented by a separate schema, instead the `ClassContents` type is used. `ClassContents` takes a `kind` attribute that defines which type it represents. Since the `ClassDefinition` type can consist of a `ClassContents` element it is possible to represent functions with a `ClassDefinition`. The same approach is used to represent enumerations and records.

Optimica

MXML is mainly intended to represent Modelica, however for this thesis a small extension with support for Optimica constructs was provided. The Optimica schema uses the already described schemas for representing the Modelica parts. The model is then extended with a `constraint` section that represents the constraints of an optimization problem. There are three different types of constraints, `equal`, `lessThan` and `greaterThan`. The start time and final time of the optimization is represented with two elements `startTime` and `finalTime`, which consists of one expression. The objective functions are represented with two elements consisting of one expression, `objective` and `objectiveIntegrand`.

3.3 Comparison of the formats

To compare the two different XML formats it is important to consider which factors that are relevant. Looking at the expressiveness of the formats they are both similar. The main difference is that FMUX is intended to only work with DAEs, which means that no discrete variables are supported[11]. Furthermore only parameter variables may be of any other type than `Real`. MXML is designed to support all constructs available in Modelica and it does not have any self imposed limitations like FMUX.

Listing 3.1: XML in MXML format to describe the Modelica model in 2.1

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <class kind="model">
```

```
3     <component name="x">
4       <builtin name="Real"/>
5     </component>
6     <component name="y">
7       <builtin name="Real"/>
8     </component>
9
10    <equation>
11      <equal>
12        <local name="x"/>
13        <call builtin="+">
14          <integer value="2"/>
15          <integer value="4"/>
16        </call>
17      </equal>
18      <equal>
19        <operator builtin="der">
20          <local name="y"/>
21        </operator>
22        <call builtin="sin">
23          <local name="x"/>
24        </call>
25      </equal>
26    </equation>
27  </class>
```

The structure of the two formats are similar overall but there are some notable differences. In MXML all information is represented by child elements and attributes. FMUX uses another approach where some of the information is kept as element content. An example of this is an integer value that MXML represents as `<integer value="2"/>`. In FMUX the representation would look like `<IntegerLiteral>2</IntegerLiteral>`. One advantage with keeping the information as an attribute is that the XML gets more concise since the element can be closed immediately.

FMUX is structured into different namespaces, which make the description of the schemas clear. However it also adds overhead to the format since the names of the elements are prefixed by the namespace. In comparison, MXML has a similar schema structure but avoids using namespaces in the XML. Since most of the constructs within the models are self explanatory the namespace prefix only adds overhead. For example consider `<equ:DynamicEquation>`, which shows how unnecessary the namespace prefix is since the element name clearly provides the needed information. There may be cases where different constructs have similar or even the same name that justifies the use of a namespace. The occurrence of this is not prevalent in the FMUX format and even if it was it would still be bad design to use ambiguous names in the schema.

The two formats have different ways of representing mathematical operators and functions. In FMUX they are represented by their name, which means that they are a part of the defining schema. In MXML they are handled as a special kind of function call. This means that instead of having special groups as the FMUX has, an optional attribute `builtin` is added to the regular function call. FMUX approach makes the XML simpler but if support for a new built-in operator/function needs to be added the schema must be extended. MXML will inherently

support all built-in functions and operators since they are not a defined part of the schema.

In FMUX each type of equation is defined by its own element in the schema. This is in contrast to MXML where the equation types are differentiated only by the attribute `kind`. Since FMUX only allows a subset of constructs in the initial equation a separate type is needed to enforce this limitation. In MXML there are no such limitations and therefore it is possible to define the type of an equation with an attribute. Another substantial difference with the representations of equations are the features supported. MXML handles all available constructs whereas FMUX only handle the function calls and equality constructs.

Both formats have good support for extensions, since FMUX was designed as an extension to a format this is natural. MXML was not designed with extensions as a priority since the purpose is to represent the Modelica language. The description of how the formats represent Optimica clearly showed that a similar approach where used for both formats. Although FMUX had a better way to integrate the extension with the other parts of the schema.

Overall the basic structure of the formats is as mentioned similar, which is not surprising since they both represent the same language with some minor restrictions on the FMUX side. However on a more detailed level the differences are significant. The comparison shows that MXML in general uses a more compressed format with fewer schema dependencies than FMUX. Listing 3.1 shows how the Modelica model in listing 2.1 could be formulated with MXML. The corresponding FMUX model is presented in listing 3.2. As seen from the example the FMUX code is more complex and harder to read. Even though readability is not an important factor since the format is mostly used by computers, it is still advantageous to have a format that is easy to understand.

Listing 3.2: XML in FMUX format to describe the Modelica model in 2.1

```
1  <jmodelicaModelDescription>
2  <ModelVariables>
3    <ScalarVariable name="x" valueReference="0" variability="constant"
        causality="internal" alias="noAlias">
4      <Real relativeQuantity="false" start="6.0" />
5      <QualifiedName>
6        <exp:QualifiedNamePart name="x"/>
7      </QualifiedName>
8      <isLinear>true</isLinear>
9      <VariableCategory>independentConstant</VariableCategory>
10   </ScalarVariable>
11   <ScalarVariable name="y" valueReference="2" variability="continuous
       " causality="internal" alias="noAlias">
12     <Real relativeQuantity="false" />
13     <QualifiedName>
14       <exp:QualifiedNamePart name="y"/>
15     </QualifiedName>
16     <isLinear>true</isLinear>
17     <VariableCategory>state</VariableCategory>
18   </ScalarVariable>
19 </ModelVariables>
20
21 <equ:DynamicEquations>
22   <equ:Equation>
```

```
23      <exp:Sub>
24        <exp:Identifier>
25          <exp:QualifiedNamePart name="x"/>
26        </exp:Identifier>
27        <exp:Add>
28          <exp:IntegerLiteral>2</exp:IntegerLiteral>
29          <exp:IntegerLiteral>4</exp:IntegerLiteral>
30        </exp:Add>
31      </exp:Sub>
32    </equ:Equation>
33    <equ:Equation>
34      <exp:Sub>
35        <exp:Der>
36          <exp:Identifier>
37            <exp:QualifiedNamePart name="y"/>
38          </exp:Identifier>
39        </exp:Der>
40        <exp:Sin>
41          <exp:Identifier>
42            <exp:QualifiedNamePart name="x"/>
43          </exp:Identifier>
44        </exp:Sin>
45      </exp:Sub>
46    </equ:Equation>
47  </equ:DynamicEquations>
48  </jmodelicaModelDescription>
```

4 Implementation

This chapter describes the implementation process of the MXML export and import. The first section provides an overview over the structure of the two subsystems and where they are located in the transfer system. After this the key concepts behind the export is explained. This is followed by a description of the import into the *CasADiInterface* and a final brief section about how the implementation was tested.

4.1 Overview

The implementation is divided into two different modules, one that handles the export of XML from models and another part that handles import of the XML into `CasADiInterface`. Both modules are built on top of already existing software. The export module is an extension to the *JModelica.org* compiler and the import module is built on top of the existing `CasADiInterface` optimization framework.

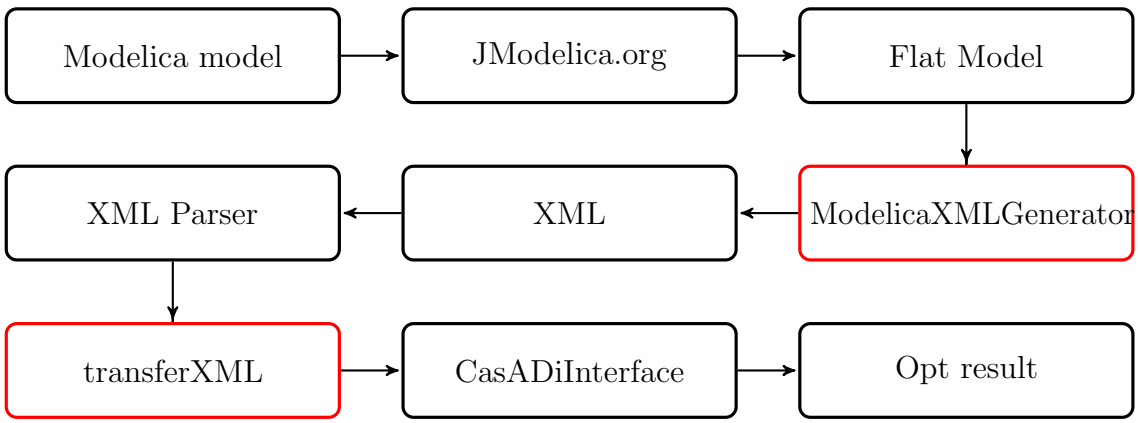

Figure 4.1: The flow of a transfer of a Modelica model from the compilation to the CasADiInterface. The red rectangles shows the parts of the transfer that have been implemented in this thesis.

Figure 4.1 presents a simplified flow diagram of how a Modelica model is transferred to the `CasadiInterface`. The flow diagram starts by showing the model being parsed into the *JModelica.org* system. As explained in section 2.5.2 several steps are performed within the compiler to obtain the flat model. The flat model is then used in the `ModelicaXMLGenerator` as basis for the XML generation. From the generated XML the import starts with parsing the XML to get a Document Object Model (DOM) to work with. The next step is to use the DOM object that was obtained from the parsing to transfer the model into the `CasADiInterface`. With the model transferred to the CasADiInterface it can be optimized and the results from

the optimization obtained. The implementation provides the `ModelicaXMLGenerator` and `transferXML` parts of the flow diagram.

4.2 Export

The export module was implemented within the *JModelica.org* compiler as a new backend service. It was constructed on top of the meta compilation tool JastAdd. The XML export takes a flat abstract syntax tree representation of a Modelica model and generates an XML representation of it. To achieve this the export module uses several other modules that are already in place in the compiler system. The printer framework, which is described in section 4.2.2, is used for convenient printing of an output file and also to handle the indentation level of the XML. For integration with the test framework and main compiler class the `GenericGenerator` framework is used, which is described in section 4.2.5.

4.2.1 Structure of the abstract syntax tree

Within *JModelica.org* the abstract syntax tree is represented with a class structure where each class represents a node in the tree. Child nodes in the tree will in the class structure be represented by subclasses. The root node of the abstract syntax tree is the `FClass` class. Each language construct will be represented with a separate node in the tree. For example variable nodes are represented with the `FVariable` class, equations with the `FEquation` class and functions with the `FFunctionDecl` class. These three nodes, together with child nodes, cover most of the constructs available in the Modelica language but there are also some other nodes that are less used 4.2 shows an overview of the abstract syntax tree structure can be seen. Note that this figure only shows the most important nodes and only a small part of their subtree.

4.2.2 Printer framework

The printer framework is used in the code generation since it provides required functionality such as printing to a stream and support for handling the indentation. The framework is structured into one class that is called `Printer` which has methods for printing to a stream object and for increasing the indentation. To support printing of nodes in the abstract syntax tree the `Printer` class has an empty implementation of a `print` method that prints an `ASTNode` object.

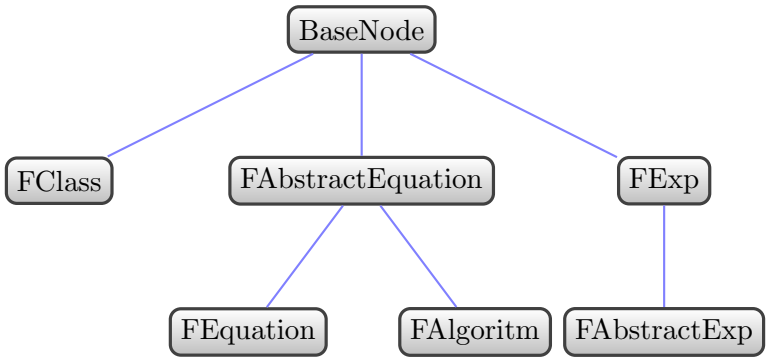

Figure 4.2: The basic structure of the abstract syntax tree that the XML generation is based on.

The framework is used by creating a subclass of the `Printer` class with an implementation of the `print` method. The code for the `Printer` subclass used in the XML generation is given in 4.1. As seen on the third line the constructor sets the step size of the indentation to one tab. On the fifth line the print method is defined and calls a method `prettyPrintXML` that is used for the generation. In section 4.2.3 it is explained how this is used as a part of the generation.

Listing 4.1: The Printer class used in the XML export

```
1 public class XMLPrettyPrint extends Printer {
2   public XMLPrettyPrint() {
3     super("\t");
4   }
5
6   public void print(ASTNode node, CodeStream str, String indent){
7     node.prettyPrintXML(this, str, indent);
8   }
9 }
```

4.2.3 XML generation

The basic idea with the XML generation is to split it into methods that generate XML for a small specific subset of the Modelica language. For example it would be convenient to have one method that generates XML for expressions, one for equations and so on for the other language constructs. The implementation is based on aspects in JastAdd that were described in section 2.4, by using aspects it is possible to have all code related to the XML generation grouped together in one file. The XML generation consists of one class `ModelicaXMLGenerator`, which implements the `prettyPrintXML` method for all nodes that should have any generation behavior. To generate XML for a node is then a simple call to the `print` method with the node passed as a parameter.

To generate XML tags a couple of helper methods are implemented that take a tag name, indentation and attributes. An example of how one of these helper methods looks can be seen in listing 4.2. The first line defines the format of the tag, in this case it is a tag that is opened and closed on the same line. There are also format strings for open and close tags. The method does two things. First it loops through all attributes and append them to a single string. Then the format tag is used to print the string to the output stream that is provided as a parameter.

Listing 4.2: Helper method for generating a XML tag to an output stream

```
1 private static final String ASTNode.CLOSED_TAG = "%s<%s%s/>\n";
2
3 public void ASTNode.generateClosedTag(CodeStream str, String tag,
4     String indent, Map<String, String> attributes) {
5     StringBuilder allAttributes = new StringBuilder();
6     if (attributes != null) {
7         for (Map.Entry<String, String> attribute : attributes.
              entrySet()) {
8             allAttributes.append(" ").append(attribute.getKey()).
                  append("=\"")
9                 .append(attribute.getValue()).append("\"");
```

```
10          }
11      }
12      str.format(CLOSED_TAG, indent, tag, allAttributes.toString());
13  }
```

A common problem with code generation is namespacing, however this is not a problem when we generate code for a model that lacks any hierarchy. Because the model is flat and does not contain any hierarchical elements, every name is guaranteed to be unique. This means that the code generation does not need any data structure for name lookup since it can always infer the correct name from the variable.

The XML generation takes an `FClass` object as basis for generation. The variables, equations and functions of a model are obtained by methods within the `FClass` object. To generate constructs on the current level in the abstract syntax tree the helper methods described earlier are used. For example consider generation of equations in the `FClass` object. Before all equations a tag `<equation>` is required. This tag is generated by calling the helper method `generateOpenTag` with the stream object, tag name and attributes. The generation of the equations is performed by calling the `print` method and passing the equation object as a parameter. The `prettyPrintXML` method for the `FEquation` object will then be called and it will generate XML for the equations. By using this approach it is easy to delegate generation of all entities in the abstract syntax tree to a separate logical part of the code.

In listing 4.3 generation code for equations with an expression as left-hand and right-hand side is presented. With the underlying structure in mind the generation is straightforward and it consists of creating the local XML tags for the node and then retrieves and delegates the generation of the expressions. The same approach is used for all nodes in the abstract syntax tree, however there are of course some nodes that are more complicated to add generation for but their generation still follow the same principle.

Listing 4.3: Code generation for equations with expressions as left-hand and right-hand sides

```
1  public void FEquation.prettyPrintXML(Printer p, CodeStream str,
       String indent){
2      String indentOneStep = p.indent(indent);
3      generateOpenTag(str, "equal", indent);
4      p.print(getLeft(), str, indentOneStep);
5      p.print(getRight(), str, indentOneStep);
6      generateCloseTag(str, "equal", indent);
7  }
```

4.2.4 Export of Optimica models

In the *JModelica.org* compiler Optimica models are represented by the `FOptClass` class as opposed to the regular `FClass` for Modelica models. `FOptClass` is an extension of `FClass` with optimization information added. Since the `FOptClass` uses the same abstraxt syntax tree classes for representing variables, equations and functions as `FClass`, most of the generation can be reused. The new constructs that must be accounted for in the Optimica generation are constraints, objective functions and start/end time for the optimization. The objective function is kept as an expression in the `FOptClass`. Constraints are represented by the `FConstraint` class, each type of constraint is a subclass to `FConstraint`. A constraint consists of a left-hand

side and right-hand side expression. The start and final time of the optimization is kept as two expressions in the `FOptClass`. Since all of these constructs are supported their generation is straightforward and follow the same pattern as the Modelica generation.

4.2.5 Integration with compiler

To use the generation it needs to be integrated with the rest of the compiler system. Before it is possible to generate XML for a Modelica model the compiler needs to flatten the model. Since the XML generator works with a flat model it must be obtained through the compiler together with an output stream that the generated XML should be written to. Similar XML generation already exists within the *JModelica.org* system and there is a framework in place to handle this problem.

JModelica.org has a `GenericXMLGenerator` class with a method `generate`, which is used as the entry point for code generation in the compiler. To integrate the generation code with the compiler a wrapper class, `ModelicaGenerator` that interfaces with the `ModelicaCompiler` is created. `ModelicaGenerator` extends `GenericXMLGenerator` and implements the `generate` method by calling `prettyPrintXML()` with the provided stream object and the `FClass` object. In `ModelicaCompiler` there are several targets defined for the different formats supported. Each target contains a generator object, which is the one used in the generation. For example the `xml` target will use the `ModelicaGenerator` whereas the `fmi` target uses the `FmiXMLGenerator`. When the target is set to `xml` the generation will generate an MXML file from the provided Modelica model.

4.3 Import

The import part of the implementation uses generated XML from the export and imports this into *CasADiInterface*. *CasADiInterface* is implemented in C++ and consists of a model description that interfaces with CasADi for solving optimization problems. The representation of a model in *CasADiInterface* consists of classes for representing variables, equations and functions as well as a model class that binds these entities together. The model class is also responsible for interactions with the users and the numerical solvers. The import takes an empty Model object and then fills it with the information from the XML document.

4.3.1 XML parsing

The first step of the import is to parse the XML document so its data can be accessed in a suitable format. There exist several different techniques for parsing XML, the two most common being Simple API for XML (SAX)[24] and DOM[25]. The main difference between SAX and DOM is how they operate on the XML document; a DOM parser will read the whole document into memory whereas a SAX parser operates on the elements in the document sequentially. In practice this means that the SAX parsers are generally much faster than DOM parsers and at the same time use less memory. However there are situations when a SAX parser is not suitable because access to the full document is required. For example validation of an XML document typically requires access to the complete document and therefore a DOM parser is better for this task.

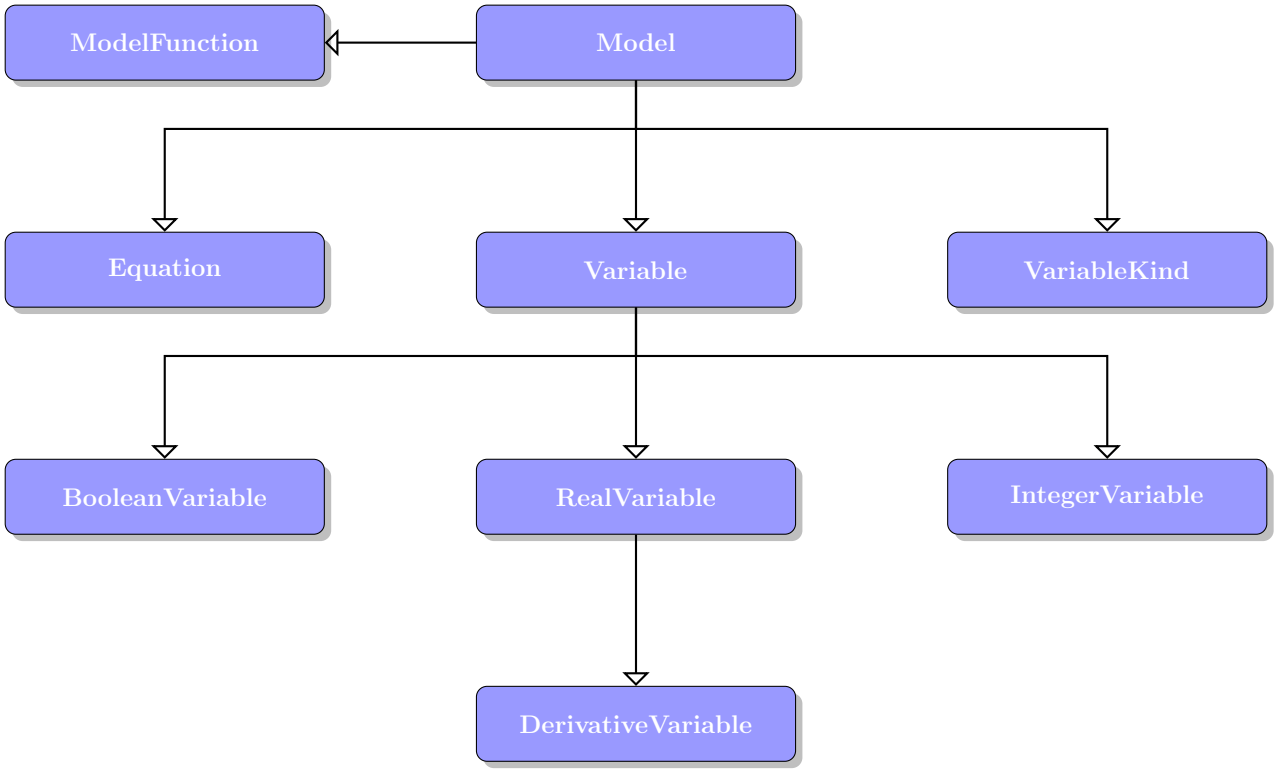

Figure 4.3: Class structure of the CasADiInterface

The import module implemented in this thesis uses a DOM parser. The reasoning behind this is that although the speed of parsing is an important factor it was more important to have a parser that is lightweight and easy to use. The parser used is *tinyxml2*, which is a simple and lightweight XML parser implemented in C++ [26]. *Tinyxml2* only consists of two files, which make it easy to integrate with the rest of the import and keeps dependencies low in the project. The basic workflow with *tinyxml2* is to first read the XML document, which gives a pointer to the root node of the DOM object. *Tinyxml2* provides methods for getting pointers to the children, siblings and parents of a node. In terms of speed *tinyxml2* is not among the fastest parsers but it can handle large documents within a reasonable amount of time.

4.3.2 CasADiInterface

The optimization toolchain *CasADiInterface* that the import works towards was developed as a part of another masters thesis[27]. The idea with *CasADiInterface* is to take a Modelica or Optimica model and convert it into a form so that it can be optimized with CasADi. Since CasADi uses an internal format for representing optimization problems the toolchain must use this representation to be able to optimize the models. The basic structure of *CasADiInterface* can be seen in figure 4.3. The `Model` class represents a complete model, with a list of variables, a list of equations and a map with the functions present in the model. The `Model` class also has methods for evaluating MX expressions and calculating the value of dependent parameters.

Variables are represented by the `Variable` class, each variable consists of a symbolic MX with the name of the variable and a map storing all the attributes of the variable. The name of the attribute is used as a key in the map and an MX variable is the value. As seen in figure

4.3 the `Variable` class has several subclasses for each different type of variable. Each subclass extends the `Variable` class with specific functionality for that type of variable, for example the `RealVariable` class is extended with a derivative variable that is not present in the regular `Variable` class. Equations are represented by the `Equation` class, which consists of one MX expression for the left-hand side and one for the right-hand side. Functions are represented by the `ModelFunction` class, which stores a `MXFunction` object that corresponds to the function and have a method for calling this function.

Optimization problems are represented by a subclass, `OptimizationProblem`, of the `Model` class, which extends it with support for the optimization constructs. The `OptimizationProblem` extends the `Model` with MX expressions for the objective function, start time and final time. Constraints are represented with an own class in which the left-hand side and right-hand side expressions are kept, together with the type of the constraint.

4.3.3 Construction of model

There are several different pieces that need to be fitted together to import a model into the CasADiInterface. To convert expressions from XML to MX a function called `expressionToMX` is implemented, which takes a reference to an expression in the XML and then construct the corresponding MX expressions. The `expressionToMx` method handles function calls, built-in expressions as well as operators. Variables must be converted to MX expression since that is how they are represented in the `Model` class. Functions and equations are also represent with the help of MX expression but not solely as one MX expression. For equations and variables this conversion is straightforward but for functions that are converted to a `MXFunction` it is more complicated.

To convert a function to an `MXFunction` object all of the input and output variables of the function need to be converted to `MX` expressions. Two lists are then created, one for the input variables and one for the output variables. For each assignment in the function the output variable is updated with the left-hand side expression of the assignment. The two lists are then provided as arguments to the `MXFunction` constructor.

4.4 Testing

Both the export and import were integrated with a small test suite to test some specific behaviour. For the export the generation was tested by comparing hand generated XML with the compiler output for a number of small models. This was used to discover if the generation is working as intended in the beginning. After this it was used to see if changes to the code did not break the current generation.

For the import a test suite was already available and it was easy to integrate with the XML based import. This test suite runs several tests that transfer a model and then compare values from the resulting `CasADiInterface` model with the correct values. Here the test was used to determine if the implementation of a feature was correct and then to assure that changes did not introduce any bugs.

5 Results

In this section the results of the implementation is described. Several tests of the functionality and performance are presented. First the tests of the functionality are explained and the results are given. In the final sections the performance tests are explained and the results are provided in several tables.

5.1 Functionality tests

The functionality of the implementation is tested in two different ways. First by two test suites, one for the export and one for the import, that test specific behavior. This is good for testing specific subsets of the implementations but to test the implementation as a whole it is not enough. To test that the implementation works as intended a model is transferred to `CasADiInterface` through XML and then optimized.

The model that is used for testing is an example model used within *JModelica.org*. The model is based on the Hicks-Ray Continuously Stirred Tank Reactors (CSTR) system. A system with two states, concentration and temperature. The control input to the system is the temperature of the cooling flow in the reactor. The model solves an optimal control problem where the objective is to ignite the reactor while avoiding uncontrolled temperature increase. In figure 5.1 the results of the optimization is presented. To verify the results they were compared with the results of the compiler transfer and this comparison showed that the result was the same for both transfers.

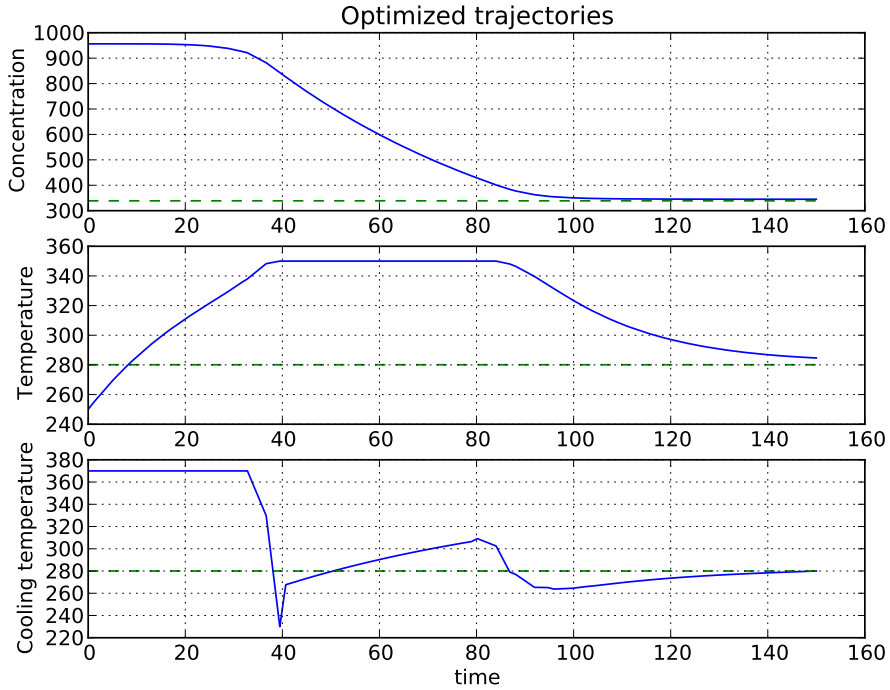

Figure 5.1: Graph showing the results of the CSTR optimization. The green dashed line indicates the desired end result whereas the blue line shows the optimization result.

Model name	Export time(avg)	Import time(avg)	Total time(avg)
VDP	3265 ms	4354 ms	7620 ms
CSTR	4818 ms	4322 ms	9141 ms
Siemens	9578 ms	4522 ms	14100 ms

Table 5.1: Time results for transfer through MXML. The results are presented in milliseconds and rounded to two decimals.

5.2 Performance tests

To test the performance of the format and the implementation two factors are considered, time and size. It is relevant to test the time of transferring a model from Modelica code to the CasADiInterface since the implementation is supposed to be used in the industry on large projects. To get some relative data on how the system performs the transfer from the compiler will be used as a comparison. The tests are conducted by timing the generation and import of a model separately, the total time is then obtained by adding the two test results together. To ensure that the test results are reliable the tests are measured several times and then the average time is presented.

Testing of the size is not so much of relevance for the implementation but rather for the format. XML is verbose by nature and it is therefore important that the format does not add substantial overhead on top of this. The format is intended to be used with large scale models that have thousands of variables, which makes it important to test that the format can handle models of that size and still keep the XML within a workable size. Another important aspect to test is how the format compares to similar formats such as `FMUX`.

The tests were conducted on three different models. The first model is the CSTR model that was used to show the functionality. The second model was another smaller example model used within the *JModelica.org* system, the purpose of this model was to optimize a Van der Pol oscillator and the model is therefore called VDP in the test results. The model code for both these models are available in appendix A. The final model is an industrial model that was developed as a part of a joint master thesis work at Siemens and Modelon[28][29]. In the test results the model is referred to as *Siemens*.

5.2.1 Time tests

In this section the results from the time tests are presented. Table 5.1 shows the results for the time tests of the XML transfer and table 5.2 shows the results for the transfer through the compiler system.

Model name	Transfer time(avg)
VDP	5534 ms
CSTR	7033 ms
Siemens	11630 ms

Table 5.2: Time results from direct transfer through the JModelica.org compiler. The results are presented in milliseconds and rounded to two decimals.

5.2.2 Size tests

The results of the size tests are presented in this section. The tables used contains the following information: the name of the tested model, the file size of the generated XML code, number of lines of the generated XML code and number of lines of the flat model file that the generation is based on. Note that the file size of the generated XML is given in *kilobytes*. In table 5.3 the results for the MXML format is presented and in table 5.4 the results for the FMUX format is presented.

Model name	XML size(kb)	Lines in XML	Lines in flat model
VDP	3	135	20
CSTR	11	434	53
Siemens	608	22389	2166

Table 5.3: This table presents the results of the size tests for the MXML format.

Model name	XML size(kb)	Lines in XML	Lines in flat model
VDP	9	258	20
CSTR	26	755	53
Siemens	5896	182429	2166

Table 5.4: This table presents the results of the size tests for the FMUX format.

6 Discussion

This section presents an evaluation of the implementation followed by an evaluation of the MXML format. A final conclusion is then given in which the fulfillment of the thesis goals are summarized. The final section presents a number of possible future extensions and continuations to the thesis.

6.1 Evaluation of implementation in JModelica.org

The implementation aimed to extend the *JModelica.org* compiler with XML generation for the MXML format and to add a module for import of MXML into *CasADiInterface*. As shown in the functionality tests in section 5.1 it is possible to transfer a model to the `CasADiInterface` through the MXML format and get an optimal solution for the optimization problem provided by the model. This result shows that the export as well as the import works as intended for small scale models that where used in the tests.

Due to lack of time some features were not implemented, this relates mostly to the import whereas the export has almost full coverage of the expected features. The main unsupported feature in the import is records, which are necessary for some problems but they are not used that often. Alias variables are currently not supported in the export and import since other parts were deemed more important to finish in the later stages of the thesis. To support all features would be an advantage but since there only are a few and not so vital parts that are unsupported it is acceptable.

The time test shows that import from the generated XML is faster than the direct transfer from the compiler. However if the generation time of the XML is included the transfer from the compiler is faster. This was expected since the generation performs almost the same work as the transfer and then the additional work of parsing the XML and constructing a DOM object is added on top of this. One important factor to consider is that the same XML file will probably be used several times. Therefore the XML generation time is less important than the test results showed and in the long term it is likely that the XML import is in fact faster.

The main purpose of the implementation was to test how suitable the MXML format was to use in a working environment. The results presented indicates that MXML is indeed a format that is easy to work with and provides the required features. The performance test results show that MXML can provide similar performance as other solutions and at the same time be more versatile. The implementation process has shown that MXML is an easy format to work with, both when it comes to generation from an abstract syntax tree and import from a generated XML document. Code generation does not necessarily occur from an abstract syntax tree so it is possible that code generation from other representations may be more complicated to implement. With regards to the import the same logic applies when parsing the XML with a non DOM parser. For example with a SAX parser problems may arise when functions need to be parsed in correct order and in the XML they are given in an arbitrary order. However this could not be considered a limitation to the MXML format but rather a problem for the implementations to solve.

6.2 Evaluation of MXML

The comparison between MXML and FMUX indicated that MXML was the more concise and extendable format. This assumption was further strengthened by the test results obtained in section 5.2.2. The tests show a substantial difference in size between the two formats. This difference applies for all the tests and for the large industrial model the difference was almost a factor of ten. Such a large difference in the size between the formats was not expected but it clearly demonstrates that MXML is less verbose and has a better structure to minimize the size of the XML.

If other aspects of the formats are considered the advantages of MXML show even clearer. It is more expressive and is designed with the ambition to represent all the features offered by Modelica. The MXML schema is also easier to extend in some regards since several properties are represented with attributes instead of with elements.

Even though MXML is an improvement from the former XML formats it can still be improved. There are some part of the schema that add complexity and sometimes they may even be redundant. To clarify or remove these parts would make the structure of the format even easier to understand and work with. Furthermore there are some disadvantages with using attributes to represent the data of the model. If Modelica changes its representation of some data it is likely that MXML must be redesigned to account for that change. It would then be better to have the data in an element since they are easier to extend than attributes are. This is however a minor issue since the format is used within a stable environment that rarely changes.

6.3 Conclusion

In conclusion the evaluation of the MXML format shows that it is a format suitable for transfer of Modelica models. The implementation of support for MXML within the *JModelica.org* system confirmed that the format was easy to integrate with an existing system. By comparing the MXML format with another XML format that is used for transfer of Modelica models the strengths of the MXML format was shown. The conducted tests confirmed the conclusions from the comparison, namely that the MXML format was superior to the FMUX format both in design and performance.

6.4 Future work

There exist several possible extensions and continuations of this thesis. One interesting extension would be to investigate whether MXML is expressive enough to represent full Modelica models and not only flat models. This would include implementation of a new backend for MXML generation of full models as well as a comparison of the flat models and the hierarchical. It would also be interesting to compare the MXML format with other types of alternative representations such as a binary model representation or a custom format. This could include design and implementation of the format and then a comparison of the formats with focus on the performance.

Apart from continued extension of MXML and comparisons with other formats it would be interesting to investigate how well the format could be used within large scale industrial projects. Since the format is developed within the MODRIO project it would be advantageous to test the format with models related to this project. One approach to test this could be to implement support for MXML within a few more Modelica tools and then conduct a study where the models where optimized or simulated from the XML representation.

References

[1] J. Åkesson. "Optimica—An Extension of Modelica Supporting Dynamic Optimization". *In 6th International Modelica Conference 2008*. Modelica Association, Mar. 2008.

[2] J. Andersson, J. Åkesson, and M. Diehl. "Dynamic optimization with CasADi". *51st IEEE Conference on Decision and Control*. Accepted for publication. Maui, Hawaii, USA, Dec. 2012.

[3] J. Åkesson, M. Gäfvert, and H. Tummescheit. "JModelica—an Open Source Platform for Optimization of Modelica Models". *Proceedings of MATHMOD 2009 - 6th Vienna International Conference on Mathematical Modelling*. Vienna, Austria: TU Wien, Feb. 2009.

[4] I. 3. *ITEA 3 · Project · 11004 MODRIO*. 2014. URL: `https://itea3.org/project/modrio.html` (visited on 06/04/2014).

[5] T. Henningson. "Modelica XML". *Modelica Design meeting, Coventry*. May 2013.

[6] A. Shitahun. *Template Based XML and Modelica Unparsers in OpenModelica*. Master's Thesis. Department of Computer and Information Science, Linköping University, Sweden, 2013.

[7] A. Shitahun et al. "Tool Demonstration Abstract: OpenModelica and CasADi for Model-Based Dynamic Optimization". *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Apr. 2013.

[8] O. Ben-Kiki, C. Evans, and I. döt Net. *YAML Ain't Markup Language (YAML™)Version 1.2 (Third Edition)*. first Edition of a Recommendation. http://yaml.org/spec/1.2/spec.pdf. 2009.

[9] E. Maler et al. *Extensible Markup Language (XML) 1.0 (Third Edition)*. first Edition of a Recommendation. http://www.w3.org/TR/2004/REC-xml-20040204. W3C, Feb. 2004.

[10] P. Fritzson and A. Pop. *Abstract ModelicaXML: A Modelica XML Representation with Applications*. 2003.

[11] R. Parrotto. *An XML Representation of DAE Systems Obtained from Continuous-Time Modelica Models*. Master's Thesis ISRN LUTFD2/TFRT--5865--SE. Department of Automatic Control, Lund University, Sweden, Nov. 2010.

[12] B. Dimic, B. Milosavljevic, and D. Surla. XML schema for UNIMARC and MARC 21. *Electronic Library* **28** (2010), 245–262.

[13] N. Nurseitov et al. "Comparison of JSON and XML Data Interchange Formats: A Case Study". *ISCA 22nd International Conference on Computer Applications in Industry and Engineering*. 2009.

[14] H. Elmqvist. "A Structured Model Language for Large Continuous Systems". PhD thesis. Department of Automatic Control, Lund University, Sweden, May 1978.

[15] *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. Version 3.2. Modelica Association, July 2013. URL: `https://www.modelica.org/documents/ModelicaSpec32Revision2.pdf`.

[16] J. Åkesson et al. Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problems. *Computers and Chemical Engineering* **34**.11 (Nov. 2010), 1737–1749.

[17] T. Parr, ed. *Language Implementation Patterns*. The Pragmatic Bookshelf, 2010.

[18] G. Hedin. "An Introductory Tutorial on JastAdd Attribute Grammars". *Generative and Transformational Techniques in Software Engineering III*. Ed. by J. Fernandes et

al. Vol. 6491. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 166–200. ISBN: 978-3-642-18022-4. DOI: 10.1007/978-3-642-18023-1_4. URL: http://dx.doi.org/10.1007/978-3-642-18023-1_4.

[19] Modelon. *JModelica.org*. 2014. URL: http://www.jmodelica.org/ (visited on 05/15/2014).

[20] modelisar. *Functional Mock-up Interface for Model Exchange and Co-Simulation*. 2012. URL: https://svn.fmi-standard.org/fmi/branches/public/specifications/FMI_for_ModelExchange_and_CoSimulation_v2.0_Beta4.pdf.

[21] F. Casella, F. Donida, and J. Åkesson. "An XML Representation of DAE Systems Obtained from Modelica Models". *Proceedings of the 7th International Modelica Conference 2009*. Modelica Association, Sept. 2009.

[22] J. Andersson. "A General-Purpose Software Framework for Dynamic Optimization". PhD thesis. Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium: Arenberg Doctoral School, KU Leuven, Oct. 2013.

[23] J. Andersson, J. Gillis, and M. Diehl. *User Documentation for CasADi v1.8.1*. 2014.

[24] D. Megginson. *SAX*. 2004. URL: http://www.saxproject.org/ (visited on 05/02/2014).

[25] W. D. IG. *W3C Document Object Model*. 2009. URL: http://www.w3.org/DOM/ (visited on 05/02/2014).

[26] L. Thomason. *tinyxml2*. 2014. URL: https://github.com/leethomason/tinyxml2 (visited on 04/26/2014).

[27] B. Lennernäs. *A CasADi based toolchain for JModelica.org*. Master's Thesis. Department of Automatic Control, Lund University, Sweden, 2013.

[28] A. Lind and E. Sällberg. *Optimization of the Start-up Procedure of a Combined Cycle Power Plant*. Master's Thesis ISRN LUTFD2/TFRT--5900--SE. Department of Automatic Control, Lund University, Sweden, June 2012.

[29] E. Sällberg et al. "Start-up Optimization of a Combined Cycle Power Plant". *9th International Modelica Conference*. Accepted for publication. Munich, Germany, Sept. 2012.

A Full code of the test models from the result section

Listing A.1: CSTR model code used for tests in result section

```
1  package CSTR
2
3  model CSTR "A␣CSTR"
4
5    parameter Modelica.SIunits.VolumeFlowRate F0=100/1000/60 "Inflow";
6    parameter Modelica.SIunits.Concentration c0=1000 "Concentration␣of␣
       inflow";
7    Modelica.Blocks.Interfaces.RealInput Tc "Cooling␣temperature";
8    parameter Modelica.SIunits.VolumeFlowRate F=100/1000/60 "Outflow";
9    parameter Modelica.SIunits.Temp_K T0 = 350;
10   parameter Modelica.SIunits.Length r = 0.219;
11   parameter Real k0 = 7.2e10/60;
12   parameter Real EdivR = 8750;
13   parameter Real U = 915.6;
14   parameter Real rho = 1000;
15   parameter Real Cp = 0.239*1000;
16   parameter Real dH = -5e4;
17   parameter Modelica.SIunits.Volume V = 100 "Reactor␣Volume";
18   parameter Modelica.SIunits.Concentration c_init = 1000;
19   parameter Modelica.SIunits.Temp_K T_init = 350;
20   Real c(start=c_init,fixed=true,nominal=c0);
21   Real T(start=T_init,fixed=true,nominal=T0);
22 equation
23   der(c) = F0*(c0-c)/V-k0*c*exp(-EdivR/T);
24   der(T) = F0*(T0-T)/V-dH/(rho*Cp)*k0*c*exp(-EdivR/T)+2*U/(r*rho*Cp)
       *(Tc-T);
25 end CSTR;
26
27 model CSTR_Init
28   extends CSTR(c(fixed=false),T(fixed=false));
29 initial equation
30   der(c) = 0;
31   der(T) = 0;
32 end CSTR_Init;
33
34 model CSTR_Init_Optimization
35
36   CSTR cstr "CSTR␣component";
37   Real cost(start=0,fixed=true);
38   Real u = Tc_ref;
39   parameter Real c_ref = 500;
40   parameter Real T_ref = 320;
41   parameter Real Tc_ref = 350;
```

```
42   parameter Real q_c = 1;
43   parameter Real q_T = 1;
44   parameter Real q_Tc = 1;
45
46 equation
47   cstr.Tc = Tc_ref;
48   der(cost) = q_c*(c_ref-cstr.c)^2 + q_T*(T_ref-cstr.T)^2 +
49                   q_Tc*(Tc_ref-cstr.Tc)^2;
50 end CSTR_Init_Optimization;
51
52 optimization CSTR_Opt2(objectiveIntegrand=1e-4*(q_c*(c_ref-cstr.c)^2
      + q_T*(T_ref-cstr.T)^2 +
53                   q_Tc*(Tc_ref-cstr.Tc)^2),
54                       startTime=0.0,
55                       finalTime=150)
56
57   input Real u(start = 350,initialGuess=350,min=230,max=370)=cstr.Tc;
58   CSTR cstr(c(initialGuess=300),T(initialGuess=300, max=350),Tc(
        initialGuess=350));
59
60   parameter Real c_ref = 500;
61   parameter Real T_ref = 320;
62   parameter Real Tc_ref = 300;
63   parameter Real q_c = 1;
64   parameter Real q_T = 1;
65   parameter Real q_Tc = 1;
66   Real q = 2*u;
67 end CSTR_Opt2;
```

Listing A.2: VDP model code used for tests in result section

```
1  model VDP
2      // State start values
3      parameter Real x1_0 = 0;
4      parameter Real x2_0 = 1;
5
6      // The states
7      Real x1(start = x1_0);
8      Real x2(start = x2_0);
9
10     // The control signal
11     input Real u;
12
13   equation
14     der(x1) = (1 - x2^2) * x1 - x2 + u;
15     der(x2) = x1;
16   end VDP;
17
18   model VDP_scaled_input
19     // State start values
```

```
20      parameter Real x1_0 = 0;
21      parameter Real x2_0 = 1;
22
23      // The states
24      Real x1(start = x1_0);
25      Real x2(start = x2_0);
26
27      // The control signal
28      input Real u(nominal=10);
29
30    equation
31      der(x1) = (1 - x2^2) * x1 - x2 + u;
32      der(x2) = x1;
33    end VDP_scaled_input;
34
35    optimization VDP_Opt2 (objectiveIntegrand = exp(p1) * (x1^2 + x2^2
         + u^2),
36                           startTime = 0,
37                           finalTime = 20)
38      parameter Real p1 = 2;
39      extends VDP(x1(fixed=true),x2(fixed=true),u(max=0.75));
40    end VDP_Opt2;
```