Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Template Based XML and Modelica Unparsers in OpenModelica

By

Alachew Shitahun

LIU-IDA/LITH-EX-A--12/046--SE

2012-08-30
Final Thesis

Template Based XML and Modelica Unparsers in OpenModelica

by

Alachew Shitahun

LIU-IDA/LITH-EX-A--12/046--SE

2012-08-30

Supervisor: Olena Rogovchenko
Examiner: Peter Fritzson
Abstract

In many areas modeling and simulation plays an important role. Currently, an equation-based object oriented modeling and simulation languages, such as Modelica, are used for multi-domain modeling, and the OpenModelica Compiler (OMC) is Modelica-based platform for modeling, compilation and simulation. In this thesis, we address two issues relevant to OMC. The first issue deals with translation from the internal equation-based model representation in OpenModelica to an Extensible Markup Language (XML) form which can be imported in to the CasADi open source toolkit, in order to enable integrated dynamic modeling and optimization. To achieve this, we have implemented a new template based XML code generator module in OMC based on an XML standard defined in OPENPROD EU project for representation of flattened Modelica model. The second issue deals with transformation of OpenModelica Abstract Syntax Tree (AST) into Modelica text (Unparser). With regard to this issue, we have implemented a new template based Modelica Unparser from OpenModelica AST into Modelica text.

Keywords: XML, Modelica XML, XML code generator, OpenModelica XML code generator, DAE Unparser
Acknowledgments

Firstly, I would like to thank and express all my gratitude to my examiner Professor Peter Fritzson, who introduced me into this research area and offering such a nice work environment, kept encouraging, and supporting my work. I am grateful to my supervisor Doctor Olena Rogovechenko for giving me valuable comments and discussions on my thesis work. I would also like to thank Mahder Gebremedhin for his help on technical questions during the process. I would also like to show my gratitude to all master’s thesis students who has been working on OpenModelica project for sharing me their experiences and joy with me. Last but not least, many thanks to my family and friends who has been supporting and encouraging me to do my best.
Contents

Chapter 1 .......................................................................................................................... 10
Introduction .................................................................................................................... 10
  1.1 Motivation ................................................................................................................ 10
  1.2 Problem Formulation ............................................................................................... 11
  1.3 Goals ........................................................................................................................ 11
  1.4 Methodology .......................................................................................................... 11
  1.5 Intended Readers .................................................................................................... 12
  1.6 Thesis Outline ....................................................................................................... 12

Chapter 2 .......................................................................................................................... 13
Background ...................................................................................................................... 13
  2.1 Modelica and OpenModelica .................................................................................. 13
  2.2 The OpenModelica Compiler Phases .................................................................... 14
  2.3 The OpenModelica Compiler Modules .................................................................. 15
  2.4 MetaModelica ........................................................................................................ 16
  2.5 The Susan Template Language .......................................................................... 17

Chapter 3 .......................................................................................................................... 19
An XML Schema for Models ....................................................................................... 19
  3.1 Model Description Schema ..................................................................................... 19
    3.1.1 FMI Extended Scalar Variables Module ......................................................... 20
    3.1.2 Expressions Module ......................................................................................... 22
    3.1.3 Functions Module ........................................................................................... 26
    3.1.4 Algorithms Module ......................................................................................... 31
    3.1.5 Equations Module ........................................................................................... 41
    3.1.6 Optimization Module ..................................................................................... 46

Chapter 4 .......................................................................................................................... 49
Implementation .............................................................................................................. 49
  4.1 Overview .................................................................................................................. 49
  4.2 XML Code Generation in OpenModelica ............................................................. 51
    4.2.1 Result Output ................................................................................................. 52
    4.2.2 The main function ......................................................................................... 52
    4.2.3 Qualified Names XML Code generation ....................................................... 53
    4.2.5 Equations XML Code generation .................................................................. 54
    4.2.6 Functions XML Code generation .................................................................. 55
List of Figures

2.1: The overall architecture of the OpenModelica environment (Taken from [8]) .......... 13
2.2: The OpenModelica compiler translation phases (Taken from [8]) ............................ 15
2.3: The OpenModelica compiler most important modules (Taken from [8]) .................. 16
2.4: AST of 10+8*3 in the language Exp ........................................................................ 17
3.1: An overall XML schema for models ........................................................................ 20
3.2: “QualifiedName” complex type ................................................................................ 21
3.3: The overall “FMIExtendedScalarvariables” schema .............................................. 21
3.4 Expression complex types schema ............................................................................. 23
3.5 "Range" element definitions ....................................................................................... 24
3.6: "RecordConstructor" complex type definition .......................................................... 25
3.7: "Function" element definition .................................................................................... 27
3.8: "FunctionVariable" complex type definition ............................................................. 28
3.9 "RecordList" element definition .................................................................................. 29
3.10: "Statement" complex type definition .................................................................... 31
3.11: "Algorithm" complex type definition ...................................................................... 32
3.12: "Algorithm" element definition ............................................................................... 32
3.13: "Assign" element definition ..................................................................................... 32
3.14: "If" element definition ............................................................................................ 33
3.15: "While" element definition ....................................................................................... 35
3.16: "For" element definition ......................................................................................... 37
3.17: "FunctionCallStatement" element definition ......................................................... 39
3.18: "Assertion" element definition ............................................................................... 40
3.19: "BindingEquation" complex type definition ........................................................... 42
3.20: "BindingEquations" element definition ................................................................... 42
3.21: "AbstractEquation" complex type definition .......................................................... 43
3.22: "InitialEquation" element definition ....................................................................... 43
3.23: "FunctionCallEquation" complex type definition ................................................... 45
3.24: "DynamicEquations" element definition ................................................................ 45
3.25: An Overall Optimization module schema definition .............................................. 47
3.26: "Constraint" complex type definition ..................................................................... 47
3.27: "TimeVariable" complex type definition ........................................................................ 48
3.28: "TimedVariable" element definition ............................................................................ 48
4.1: The process of compiling Modelica model to XML code in OMC ................................. 49
List of Tables

3.1 "exp" namespace complex types ................................................................. 22
3.2: "exp" namespace elements ........................................................................ 24
3.3: "fun" namespace complex types and elements ............................................ 26
3.4: "Alg" namespace complex types and elements ............................................ 31
3.5: "equ" namespace complex types and elements .......................................... 41
3.6: "opt" namespace complex types and element ............................................ 46
List of Listings

2.1: Exp Abstract syntax definition using MetaModelica Uniontype constructs .......................... 16
2.2: Template “exp” - Pattern matching example ........................................................................... 17
2.3: Template “programmerList( )” - Iterator expression example ............................................. 18
2.4: Template call example ............................................................................................................. 18
2.5: Output of Listing 2.4 .................................................................................................................. 18
4.1: SimCode Uniontype data structure .......................................................................................... 50
4.2: ModelInfo Uniontype data structure ....................................................................................... 51
4.3: SimVars Uniontype data structure ............................................................................................ 51
4.4 Top level template for XML code generation .............................................................................. 52
4.5: The main template for XML code generation ............................................................................ 53
4.6: Parts of code for qualified name of variables XML code generation ..................................... 53
4.7: Parts of code for array subscripts of qualified name XML code generation ......................... 54
4.8: Root template for Initial equations XML code generation ...................................................... 55
4.9: Root template for functions XML code generation ................................................................... 55
4.10: Root template for Algorithms XML code generation .............................................................. 56
4.11: Root template for Records XML code generation .................................................................. 57
4.12: compWithSplitElements Uniontype data structure ............................................................... 57
4.13: splitElements Uniontype data structure ............................................................................... 58
4.14: functionList Uniontype data structure .................................................................................. 58
4.15: dumpDAE template-top level template for DAE Unparser ....................................................... 58
4.16: dumpComp and dumpCompStream templates to unparse components and elements ... 59
4.17: dumpFunctions and dumpFunction templates to unparse functions ...................................... 60
5.1: The Modelica code for the recordVariability model ................................................................. 61
5.2: The flattened Modelica code for the recordVariability model ................................................. 62
5.3: The Modelica code for the Cross model ................................................................................... 63
5.4: The flattened Modelica code for the Cross model .................................................................... 65
5.5: The Modelica code for the VDP_Opt model ............................................................................. 65
5.6: The flattened Modelica code for the VDP_Opt model ............................................................. 66
Acronyms

**AST**  Abstract Syntax Tree

**XML**  Extensible Markup Language

**OMC**  OpenModelica Compiler

**DAE**  Differential Algebraic Equation

**FMI**  Functional Mockup Interface

**OMEdit**  OpenModelica Connection Editor
Chapter 1

Introduction

This chapter provides an overview to the research work presented in this thesis. It explains the motivation for pursuing this work and presents the goals. In addition it provides an overview of the methodology we used to achieve our goals. Finally, it introduces the organization of the thesis.

1.1 Motivation

Object oriented modeling plays an important role in mathematical and equation based modeling and simulation. At present, Modelica [1] is a non-proprietary, mathematical and equation based, object oriented modeling and simulation language which is used for multi-domain modeling such as mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications and generation and distribution of electric power.

Modelica has a fast growing code base and users which needs tools for different tasks such as Analysis of Modelica programs (model checkers and validators), translation between Modelica and other Modeling languages (interchange), and query of models. To perform all these tasks an alternative structured representation which should be easy accessible from any programming language, support validation, and easy to transform, query and manipulate is needed. An Extensible Markup Language (XML) has all these properties.

An Unparser is a program which transforms OpenModelica AST into modelica text. Currently the available Unparser in OMC is hand written in MetaModelica that transforms OpenModelica AST into text. Even though this hand written Unparser has good performance, still it has several shortcomings such as lack of extensibility and modeling capability, clustering of codes with mixture of conditional print statement and program logic, and code reusability problem [2]. In order to overcome these problems, a text template language has been developed recently for OpenModelica. A text template language, Susan, is strongly typed and expression oriented language designed for OpenModelica [2]. Thus, to take an advantage of text template language and standardized XML format for Modelica, it became
important to develop a new Modelica and XML Unparser in OpenModelica using the OpenModelica text template language, Susan.

**1.2 Problem Formulation**

Recently an XML schema for representation of expanded Modelica model was introduced as a part of OPENPROD EU project in [3] [4]. It was designed as standardized format for model storage and exchange which can also be imported in the CasADi open source toolkit, in order to enable integrated dynamic modeling and optimization. However, the implementation of XML code generation according to this schema is not done yet in OpenModelica. Thus the first problem is that,

**Problem 1:** Given an XML schema standard defined in OPENPROD EU project, how to implement a template based XML code generation module in OpenModelica according to the XML Schema?

An Unparser is a program that transforms abstract syntax into text. Currently there exists hand-written Unparser for OpenModelica AST into Modelica text. Even though this hand written Unparser has good performance, still it has several shortcomings such as lack of extensibility, clustering of codes with mixture of conditional print statement, and code reusability problem [2]. So, the second problem to be tackled in this thesis is:

**Problem 2:** How to implement a more concise and readable Unparser for OpenModelica AST into Modelica text using the text template language?

**1.3 Goals**

The main goals of this thesis work are the following:

- To develop template based XML code generator in OpenModelica according to an XML standard defined in the OPENPROD EU project available in [3, 4].
- To develop template based Unparser from OpenModelica AST to Modelica text.
- To perform example test cases that demonstrates the possibility of OMC to use an XML representation to export Modelica models.

**1.4 Methodology**

The methodology used for the development of Template Based XML and Modelica Unparser for OpenModelica is based upon:

**Literature Study** – A literature study is the base for the initiation of this project. In the process of understanding the OpenModelica project and this thesis work, we studied the
scientific publications and documentations on OpenModelica available in [5]. We also studied MetaModelica courses for developer available in [5] for learning the MetaModelica language.

**Implementation** – The implementation leads to obtain a deeper understanding about the research problems and its proposed solutions. We developed our template based XML and Modelica Unparser using the OpenModelica text template language, Susan, in the OpenModelica Eclipse plug-in Modelica Development Tooling (MDT) and debugger.

### 1.5 Intended Readers
This thesis work is intended for reader’s familiar with Modelica language construct, OpenModelica modeling and simulation environment, and an XML schema representation. However, interested readers with basic knowledge of compiler construction and programming can understand it too.

### 1.6 Thesis Outline
The rest of this thesis is organized as follows.

**Chapter 2:** Background information on the OpenModelica and compiler construction.

**Chapter 3:** Presents an XML schema designed for models.

**Chapter 4:** Presents the implementation of Template Based XML and Modelica Unparser in OpenModelica.

**Chapter 5:** Presents a test case that demonstrates the applicability of the implementation.

**Chapter 6:** Concludes the thesis work and gives the possible directions for the future work.
Chapter 2

Background

In this chapter, the theoretical background needed to understand the implementation is presented. The first section provides an overview of Modelica and OpenModelica environment. The second section presents an introduction to compiler construction. The OMC phases and modules are presented in section three and four respectively. Finally, it introduces the Susan template language which will be used for the implementation of this thesis work.

2.1 Modelica and OpenModelica

Modelica is an object oriented and equation based language for modeling and simulation of multi domains such as mechanical, electrical, hydraulic, control subsystems etc. [1]. It is developed by Modelica Association [6].

OpenModelica is an open source project for Modelica based modeling, compilation and simulation environment [5]. It was started at Linköping University, PELAB and its development is supported by the Open Source Modelica Consortium (OSMC) [7]. The current version of the OpenModelica environment includes different tools as shown below in figure 2-1.

![Figure 2.1: The overall architecture of the OpenModelica environment (Taken from [8])](image)

- A Modelica Compiler - The OMC translates Modelica source code to executing simulation.
• An interactive session handler (OMShell) – An OMShell provides a command interface to OMC.
• The OpenModelica Notebook (OMNotebook) – OMNotebook gives tutorial for Modelica. In addition, Modelica models can be written and simulated on it.
• The OpenModelica Development Environment (OMDev) – The OMDev is a tool for building OMC.
• A Modelica eclipse plugin (MDT-Modelica Development Tooling) – MDT is used for Modelica development, code browsing, and simulation.
• Modelica Debugger – The Modelica debugger used for debugging an extended algorithm subset of Modelica by using eclipse for displaying and positioning.
• Graphical model editor – The OpenModelica Connection Editor (OMEdit) used for graphical model editing, plotting and browsing of the Modelica standard library.

2.2 The OpenModelica Compiler Phases
In this section, we provide a short overview of OMC translation stages from Modelica source code to executing simulation which can be seen as figure 2.2 below.

In the first phase of OMC the Modelica parser produce AST from the Modelica source code. In the second phase a translator converts the AST into a flattened model with list of variables, equations, and functions. Type checking, import statements, handling of inheritance, modifications and all other object oriented operations are also performed in this phase. After the equations have been flattened, the equation analyzer and optimizer compile the model. In the last phase of OMC, the code generator generates C code which is then pass to a C compiler to produce executable code for simulation.
2.3 The OpenModelica Compiler Modules
The OMC has around 40 modules. The brief descriptions of each module can be found in OpenModelica system documentation [8]. However, in this section we will describe only the most important modules of OMC, see also figure 2.3 below.

1. The parser generates Abstract Syntax (Absyn) which is then converted into simplified intermediate form (SCode).
2. The code instantiation module (Inst) calls Lookup to find a name in an environment. It also generates the Differential Algebraic Equation (DAE) representation which is simplified by DAELow.
3. The Ceval module performs compile time or interactive expression evaluation and returns values.
4. The static module performs static semantic and type checking.
5. SimCode is the data structure for representing solved equation code which then can be used to generate different target code in code generation phase.
2.4 MetaModelica

MetaModelica is an extended subset of Modelica used for the development of OMC. The main MetaModelica language constructs extended from Modelica that are described in [9][10] are `matchcontinue`, `uniontype`, `list` and the `option` type.

The MetaModelica `uniontype` construct used to specify the type of each node in the AST. It declares one or more record members. The structure of `uniontype` may be recursive (i.e., the records are allowed to contain `uniontype` members). See Listing 2.1 for an example on how to declare an expression `uniontype` of six record types.

```plaintext
uniontype Exp
    record INT Integer value; end INT;
    record ADD Exp lhs; Exp rhs; end ADD;
    record SUB Exp lhs; Exp rhs; end SUB;
    record MUL Exp lhs; Exp rhs; end MUL;
    record DIV Exp lhs; Exp rhs; end DIV;
    record NEG Exp exp1; end NEG;
end Exp;
```

Listing 2.1: Exp Abstract syntax definition using MetaModelica Uniontype constructs
The AST representation of the expression 10+3*8 by using the \texttt{Exp} abstract syntax definition is shown below in Figure 2-4.

![Figure 2.4: AST of 10+8*3 in the language Exp](image)

2.5 The Susan Template Language
Susan is a functional, strongly typed, expression oriented template language which is specifically designed for OMC with the following main advantages [2] [9]:

- Increases readability
- Compiled to reach maximum performance
- Provides a full vehicle for different target code generation such as C++, Java, C# etc...

Susan template expressions can consists of conditional expressions, match-expressions, function calls, iterator expressions, and etc. For complete description of the Susan template expressions we refer to [9]. The match expression in the template language is used for distinction of AST structure nodes declared in MetaModelica union types. See Listing 2.2 for an example of how pattern matching against a tree structure of figure 2.4 is written in Susan template language where the type \texttt{Exp} is declared in MetaModelica \texttt{uniontype} constructs shown above in listing 2.1.

```
template exp(Exp inExp) ::= 
  match inExp  
    case INT(____) then  value 
    case ADD(____) then  ' <%exp(lhs)% > + <%exp(rhs)% > ' 
    case SUB(____) then  ' <%exp(lhs)% > - <%exp(rhs)% > ' 
    case MULT(____) then  ' <%exp(lhs)% > * <%exp(rhs)% > ' 
    case DIV(____) then  ' <%exp(lhs)% > / <%exp(rhs)% > ' 
    case NEG(____) then  ' - <%exp(exp1)% > ' 
  end exp;
```

Listing 2.2: Template “exp” - Pattern matching example
In listing 2.2, the template function \texttt{exp} is recursive. The scope of the \texttt{INT} constructor is automatically opened by using \texttt{INT(__)} pattern to make its field (i.e., \texttt{value}) available, and the \texttt{ADD, SUB, MUL, DIV, and NEG} constructors are opened automatically by using the \texttt{ADD(__)}, \texttt{SUB(__)}, \texttt{MUL(__)}, \texttt{DIV(__)}, and \texttt{NEG(__)} pattern respectively.

The iterator expressions can be seen as array or list in functional language. See Listing 2.3 for an example on how iterator expression can be used in Susan template language.

```
template programmerList(list<String> programmers)
::=
  if programmers then
  <<
    Hello <%= (programmers |> programmer => <%= programmer %>)
    ;separator="","%>
  >>
  else 'No programmer list is found'
end programmerList;
```

Listing 2.3: Template “programmerList()” - Iterator expression example

The template functions in Susan are called in the same way as functions in c (i.e. \texttt{functionName (arg1, arg2, …,argN)}), See also listing 2.4 below for an example on how to call template function defined in listing 2.4 above.

```
programmerList({'Peter, Olena, Alachew})
```

Listing 2.4: Template call example

The output of the template function call \texttt{programmerList ('Peter, Olena, Alachew')} is:

```
Hello Peter, Olena, Alachew
```

Listing 2.5: Output of Listing 2.4
Chapter 3

An XML Schema for Models

This chapter provides a detail description of an XML schema defined in [3] [4] for exchange of models.

3.1 Model Description Schema

All information related to a model including variables, variable declarations, equations, record declarations, algorithms, functions, objective functions, constraints are stored in XML format. The structure of all such XML files is defined in [3] [4] with the schema file “FmiExtended Model Description.xsd”.

The overall XML schema file is composed of different modules and result in a unified schema for models as shown below in figure 3.1.

The different modules are:

- The FMI extended scalar variables module
- The expressions module
- The functions module
- The algorithms module
- The equations module and
- The Optimization module
The detail description of each module is described in the next section.

3.1.1 FMI Extended Scalar Variables Module
The FMI XML schema is a result of ITEA2 project MODELISAR with a goal of dynamic system models defined by differential, algebraic and discrete equations of different software system can be used together for simulation [11]. The FMI extended scalar variable module [3][4] is an extension of the FMI XML schema. It uses qualified name representation of scalar variables as extension of the FMI XML schema.

Complex type “QualifiedName” for standard representation of variable identifiers is defined as:
Figure 3.2: “Qualified Name” complex type

It consists of an attribute “Name” to represent the variable identifier name, and an optional “ArraySubscripts” element to represent each index of the array as expression.

Figure 3.3: The overall “FMIExtendedScalarVariables” schema

**Example1.** Representation of a variable qualified name

Given a variable x of array type declaration (e.g., parameter Integer x [1, 1] in Modelica language), the resulting XML representation valid according to the XML schema is:

```
<ScalarVariable name="x[1,1]" valueReference="2" variability="parameter"
  causality="internal" alias="noAlias">
  <Integer start="0.0" fixed="true"/>
  <QualifiedName>
    <exp:QualifiedNamePart name="x">
      <exp:ArraySubscripts>
       
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
  </QualifiedName>
</ScalarVariable>
```

<exp:QualifiedNamePart name="x">
      <exp:ArraySubscripts>
       
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
3.1.2 Expressions Module

In “FmiExtendedModelDescription.xsd” schema, all the expressions of models written in Modelica are collected under the “exp” namespace. The “exp” namespace includes the following complex types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp</td>
<td>Base expression</td>
</tr>
<tr>
<td>UnaryOperation</td>
<td>Unary operation—operations having one operand</td>
</tr>
<tr>
<td>BinaryOperation</td>
<td>Binary operation—operations having two operand</td>
</tr>
<tr>
<td>BuiltInFunct</td>
<td>Built-in mathematical function type with one argument</td>
</tr>
<tr>
<td>BuiltIn2Funct</td>
<td>Built-in mathematical function type with two arguments</td>
</tr>
<tr>
<td>BuiltIn1or2Funct</td>
<td>Built-in function type with one argument and another optional one</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>User-defined function call type</td>
</tr>
<tr>
<td>Array</td>
<td>Array data structure</td>
</tr>
<tr>
<td>RecordConstructor</td>
<td>Constructor function of a record</td>
</tr>
<tr>
<td>QualifiedName</td>
<td>Structured variable identifier representation(see also section 3.11)</td>
</tr>
</tbody>
</table>

Table 3.1 "exp" namespace complex types
Figure 3.4 Expression complex types schema
<table>
<thead>
<tr>
<th>Elements</th>
<th>Complex type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add, Sub, Mul, Div, Pow</td>
<td>BinaryOperation</td>
<td>Binary addition, subtraction, multiplication, division, power operation</td>
</tr>
<tr>
<td>Neg</td>
<td>UnaryOperation</td>
<td>Unary negation expression</td>
</tr>
<tr>
<td>LogLt, LogLeq, LogGt, LogGeq, LogEq, LogNeq</td>
<td>BinaryOperation</td>
<td>Logical expression less than, less than or equal, greater than, greater than or equal, equal, not equal</td>
</tr>
<tr>
<td>Not</td>
<td>UnaryOperation</td>
<td>Not expression</td>
</tr>
<tr>
<td>And, Or</td>
<td>BinaryOperation</td>
<td>Logical And and Or expression</td>
</tr>
<tr>
<td>RealLiteral</td>
<td>real</td>
<td>Real literal expression</td>
</tr>
<tr>
<td>IntegerLiteral</td>
<td>integer</td>
<td>Integer literal expression</td>
</tr>
<tr>
<td>BooleanLiteral</td>
<td>boolean</td>
<td>Boolean literal expression</td>
</tr>
<tr>
<td>StringLiteral</td>
<td>string</td>
<td>String literal expression</td>
</tr>
<tr>
<td>Der, Sin, Cos, Tan, Asin, Acos, Atan, Sinh, Cosh, Tanh, Exp, Log, Log10, Abs, Sign, Sqrt, NoEvent</td>
<td>BuiltInFunct</td>
<td>Built-in derivative, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, exponential, log, log10, abs, sign, sqrt, NoEvent functions</td>
</tr>
<tr>
<td>Atan2</td>
<td>BuiltIn2Funct</td>
<td>Built-in atan2 function</td>
</tr>
<tr>
<td>Min, Max</td>
<td>BuiltIn1or2Funct</td>
<td>Built-in min(x,y) and max(x,y) function where x and y must be a scalar</td>
</tr>
<tr>
<td>Identifier</td>
<td>QualifiedName</td>
<td>Variable identifier</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>FunctionCall</td>
<td>User-defined function call expression</td>
</tr>
<tr>
<td>Time</td>
<td>none</td>
<td>Time variable</td>
</tr>
<tr>
<td>Range</td>
<td>none</td>
<td>Expression representing values in range</td>
</tr>
<tr>
<td>UndefinedDimension</td>
<td>none</td>
<td>Expressions to be used in array definition, for undefined size</td>
</tr>
</tbody>
</table>

Table 3.2: "exp" namespace elements

Element “Range” of expression module is defined as:

![Figure 3.5 "Range" element definitions](image-url)
It represents an interval of values in for loops and array constructor. The “Range” element consists of two or three scalar expressions defined as follows:

- Two scalar expressions representing the lower and upper bounds of the range
- Three scalar expressions representing the lower bound, the step size of the loop and upper bound of the range.

**Example 2.** Representation of expressions

Given an expression (e.g., \((1 - x^2)^2 \times x1 - x2 + u\) in Modelica language), the resulting XML representation valid according to the XML schema is:

```xml
<exp:Add>
    <exp:Mul>
        <exp:Sub>
            <exp:RealLiteral>1.0</exp:RealLiteral>
            <exp:Pow>
                <exp:Identifier>
                    <exp:QualifiedNamePart name="x2"/>
                </exp:Identifier>
                <exp:RealLiteral>2.0</exp:RealLiteral>
            </exp:Pow>
            <exp:Identifier>
                <exp:QualifiedNamePart name="x1"/>
            </exp:Identifier>
        </exp:Sub>
    </exp:Mul>
</exp:Add>
```

Figure 3.6: "RecordConstructor" complex type definition
3.1.3 Functions Module

In “FmiExtendedModelDescription.xsd” schema, the function sections of models written in Modelica are represented under the “fun” namespace. The “fun” namespace includes the following complex types and elements:

<table>
<thead>
<tr>
<th>Complex Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionVariable</td>
<td>Variable used by the function</td>
<td></td>
</tr>
<tr>
<td>RecordVariable</td>
<td>Declaration of a record variable</td>
<td></td>
</tr>
<tr>
<td>DerivativeInputVariable</td>
<td>Input variable used by derivative functions</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Function definition with embedded XML algorithm code</td>
<td></td>
</tr>
<tr>
<td>FunctionsList</td>
<td>List of user-defined functions</td>
<td></td>
</tr>
<tr>
<td>Recordslist</td>
<td>List of record declarations</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: "fun" namespace complex types and elements

The “Function” element of the function module under the “fun” namespace is defined as show below in figure 3.7:
The “Function” element represents all the function definitions in the model such as the name of the function, input variables, output variables, protected variables, algorithm, and, optionally, the derivate and inverse functions definition.

In the function module, there are also elements such as “InputVariable”, “OutputVariable”, and “ProtectedVariable” defined as “FunctionVariable” complex type, see also figure 3.8 below.

The “FunctionVariable” complex type of the function module is defined as:
Figure 3.8: "FunctionVariable" complex type definition

It consists of two attributes: "Type" attribute with valid values of "Integer", "Boolean", "Real", "Record", and "String" and an optional "Variability" attribute with valid values of "parameter", "constant", and "continues". Moreover, it consists of element "Name" defined as "QualifiedName" complex type and an optional "Record", "BindingExpression", and "Size" elements. If the value of "Type" attribute for variable is "Record", the "Record" element defined as "QualifiedName" complex type should be used. The element "BindingExpression" represents the initial value of the variable. The "Size" element represents array size index expression and should be used if the variable is an array.

The "RecordList" element of the function module represents all the records definition where each record is represented in a different "Record" element is defined as:
The “Record” elements are defined as “RecordVariable” complex type which consists of a “Name” element defined as “QualifiedName” complex type, which represent the name of the record and a list of “Field” elements defined as “FunctionVariable” complex type.

**Example3.** Representation of records and functions definition

```plaintext
record R
  Real x;
  Real y;
end R;
function f
  input R r;
  output Real y;
end f;
```
Given the above record and function definition in Modelica language, the resulting XML representation valid according to the XML schema is:

```xml
<fun:RecordsList>
  <fun:Record>
    <fun:Name>
      <exp:QualifiedNamePart name='R'/>
    </fun:Name>
    <fun:Field type="real">
      <fun:Name>
        <exp:QualifiedNamePart name="x"/>
      </fun:Name>
    </fun:Field>
    <fun:Field type="real">
      <fun:Name>
        <exp:QualifiedNamePart name="y"/>
      </fun:Name>
    </fun:Field>
  </fun:Record>
</fun:RecordsList>

<fun:FunctionList>
  <fun:Function>
    <fun:Name>
      <exp:QualifiedNamePart name="f"/>
    </fun:Name>
    <fun:OutputVariable type="real">
      <fun:Name>
        <exp:QualifiedNamePart name="y"/>
      </fun:Name>
    </fun:OutputVariable>
    <fun:InputVariable type="Record">
      <fun:Name>
        <exp:QualifiedNamePart name="r"/>
      </fun:Name>
      <fun:Record>
        <exp:QualifiedNamePart name="R"/>
      </fun:Record>
    </fun:InputVariable>
  </fun:Function>
</fun:FunctionList>
```
3.1.4 Algorithms Module

In “FmiExtendedModelDescription.xsd” schema, the algorithm descriptions of models written in Modelica are represented under the “fun” namespace but defined in a different module of the schema than the “function” element. This module includes the following complex types and elements as shown below in the table 3.4.

<table>
<thead>
<tr>
<th>Complex Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Type</td>
<td>Statement</td>
<td>Base statement</td>
</tr>
<tr>
<td>Complex Type</td>
<td>ConditionalStatment</td>
<td>Statement with a condition (if, while, when...)</td>
</tr>
<tr>
<td>Complex Type</td>
<td>IterationIndex</td>
<td>An index used by iteration statements</td>
</tr>
<tr>
<td>Complex Type</td>
<td>FunctionCallLeft</td>
<td>Element of the left part of a function call equation</td>
</tr>
<tr>
<td>Complex Type</td>
<td>Algorithm</td>
<td>Algorithm complex type</td>
</tr>
<tr>
<td>Element</td>
<td>Algorithm</td>
<td>Algorithm element</td>
</tr>
<tr>
<td>Element</td>
<td>Assign</td>
<td>Assign statement</td>
</tr>
<tr>
<td>Element</td>
<td>Break</td>
<td>Break statement (used in While and For statements only)</td>
</tr>
<tr>
<td>Element</td>
<td>Return</td>
<td>Return statement</td>
</tr>
<tr>
<td>Element</td>
<td>If</td>
<td>IF statement with ElseIf and Else branches</td>
</tr>
<tr>
<td>Element</td>
<td>While</td>
<td>While loop statement</td>
</tr>
<tr>
<td>Element</td>
<td>For</td>
<td>For loop statement</td>
</tr>
<tr>
<td>Element</td>
<td>FunctionCallSatatment</td>
<td>Function call statement</td>
</tr>
<tr>
<td>Element</td>
<td>Assertion</td>
<td>Assertion statement</td>
</tr>
</tbody>
</table>

Table 3.4: "Alg" namespace complex types and elements

Complex type “Statement” of algorithms module is defined as:

![Figure 3.10: "Statement" complex type definition](image)
An algorithm statement can be a simple assignment, a loop, or conditional statement.

**Complex type “Algorithm” of algorithms module is defined as:**

Figure 3.11: "Algorithm" complex type definition

**Element “Algorithm” of algorithms module is defined as:**

Figure 3.12: "Algorithm" element definition

**Element “Assign” of algorithms module is defined as:**

Figure 3.13: "Assign" element definition

It represents an assignment statement. It is defined by a variable identifier and an expression representing the new value that should be assigned to the variable.

**Example4. Representation of an assignment statement**

Given an assignment statement (e.g., `a = a + 0.5` in Modelica language), the resulting XML representation valid according to the XML schema is:

```xml
<fun:Assign>
  <exp:Identifier>
    <exp:QualifiedNamePart name="a"/>
  </exp:Identifier>
</fun:Assign>
```
Element “If” of algorithms module is defined as:

![Diagram of If statement]

*Figure 3.14: "If" element definition*

It represents “if” statements with an optional “ElseIf” and “Else” statements. All this elements are defined as “ConditionalStatement” complex type.

**Example 5.** Representation of an if statement

```plaintext
if i < 1 then
    x := 1.0;
elsif i < 2 then
    x := 2.0;
else
    x := 3.0;
```
end if;

Given the above "if" statement in Modelica language, the resulting XML representation valid according to the XML schema is:

```xml
<fun:If>
  <fun:Condition>
    <exp:LogLt>
      <exp:Identifier>
        <exp:QualifiedNamePart name="i"/>
      </exp:Identifier>
      <exp:IntegerLiteral>1</exp:IntegerLiteral>
    </exp:LogLt>
  </fun:Condition>
  <fun:Statments>
    <fun:Assign>
      <exp:Identifier>
        <exp:QualifiedNamePart name="x"/>
      </exp:Identifier>
      <fun:Expression>
        <exp:RealLiteral>1.0</exp:RealLiteral>
      </fun:Expression>
    </fun:Assign>
  </fun:Statments>
  <fun:ElseIf>
    <fun:Condition>
      <exp:LogLt>
        <exp:Identifier>
          <exp:QualifiedNamePart name="i"/>
        </exp:Identifier>
        <exp:IntegerLiteral>2</exp:IntegerLiteral>
      </exp:LogLt>
    </fun:Condition>
    <fun:Assign>
      <exp:Identifier>
        <exp:QualifiedNamePart name="x"/>
      </exp:Identifier>
      <fun:Expression>
        <exp:RealLiteral>1.0</exp:RealLiteral>
      </fun:Expression>
    </fun:Assign>
  </fun:ElseIf>
</fun:If>
```
Element “While” of algorithms module is defined as:

It is defined as “ConditionalStatement” complex type which has two child elements used to represent a condition and a list of statements inside a while conditional loops.

**Example 6.** Representation of while statement

**While noEvent**(delta >= eps) **loop**

\[
\text{sum} := \text{sum} + \text{delta};
\]

**end while** ;

Given the above “While-loop” statement in Modelica language, the resulting XML representation valid according to the XML schema is:

\[
<\text{fun:While}>
\]

\[
<\text{fun:Condition}>
\]
Element “For” of algorithms module is defined as:
Figure 3.16: "For" element definition

It is defined by two elements: "Index" element, that represents "For" statements for iteration loops and "Statements" element, that represents list of statements inside the "For" loop. The "Index" element is defined as "IterationIndex" complex type which defines the iteration variable and the iteration set. The "IterationSet" element which is defined as an expression is either an "Array" or "Range" elements as described in section 3.1.2.

Example 7. Representation of for-loop statement

\[
\text{for } i \text{ in } 1:2:3 \text{ loop } \\
\quad a[i+1] := a[i] + 1.0; \\
\text{end for;}
\]

Given the above "For-loop" statement in Modelica language, the resulting XML representation valid according to the XML schema is:

\[
<\text{fun:For}>
\text{fun:Index}>
\text{fun:IterationVariable}>
\text{exp:QualifiedNamePart name="i"}>
\text{/fun:IterationVariable}>
\text{fun:IterationSet}>
\text{fun:Range}>
\text{exp:IntegerLiteral}1</exp:IntegerLiteral>
\text{exp:IntegerLiteral}2</exp:IntegerLiteral>
\text{exp:IntegerLiteral}3</exp:IntegerLiteral>
\text{/exp:Range}>
\text{/fun:IterationSet}>
\text{/fun:For}>
\]
<fun:Index>
<fun:Statments>
<fun:Assign>
<exp:Identifier>
<exp:QualifiedNamePart name="a">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:Add>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
<exp:Identifier>
<exp:QualifiedNamePart name="i"/>
</exp:Identifier>
</exp:Add>
</exp:IndexExpression>
</exp:ArraySubscripts>
<exp:QualifiedNamePart>
</exp:Identifier>
<fun:Expression>
<exp:Add>
<exp:RealLiteral>1.0</exp:RealLiteral>
<exp:Identifier>
<exp:QualifiedNamePart name="a">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:Identifier>
<exp:QualifiedNamePart name="i"/>
</exp:Identifier>
</exp:IndexExpression>
</exp:ArraySubscripts>
<exp:QualifiedNamePart>
</exp:Identifier>
</exp:Add>
</fun:Expression>
</fun:Assign>
</fun:Statments>
</fun:For>
Element “FunctionCallStatement” of algorithms module is defined as:

It represents an assignment statement where the right hand side of the statement is a user-defined function call that returns multiple arguments, possibly empty arguments.

**Example 8.** Representation of function call statement

Given a function call statement (e.g., \((y, z) := F(x)\) where \(F\) is a user defined function in Modelica language), the resulting XML representation valid according to the XML schema is:

```xml
<fun:FunctionCallStatement>
    <fun:OutputArgument>
        <exp:Identifier>
            <exp:QualifiedNamePart name="y"/>
        </exp:Identifier>
        <exp:Identifier>
            <exp:QualifiedNamePart name="z"/>
        </exp:Identifier>
    </fun:OutputArgument>
    <exp:FunctionCall>

```
Element “Assertion” of algorithms module is defined as:

Figure 3.18: "Assertion" element definition

It represents assertions of the algorithms by defining “Message” element which is optional, a “Condition” element which is a Boolean expression, and an optional “Level” attribute with a value of “error” or “warning”.

**Example 9.** Representation of an assertion statement

Given an assert statement in Modelica language (e.g., `assert (assertTest.x >= assertTest.lowlimit AND assertTest.x <= assertTest.highlimit,"Variable x out of limit");`) where `assertTest.x = 5.0`, the resulting XML representation valid according to the XML schema is:

```
<fun:Assertion>
  <fun:Condition>
    <exp:And>
      <exp:LogGeq>
        <exp:RealLiteral>5.0</exp:RealLiteral>
        <exp:Identifier>
          <exp:QualifiedNamePart name="x"/>
        </exp:Identifier>
      </exp:LogGeq>
      <exp:And>
        <exp:LogLeq>
          <exp:Identifier>
            <exp:QualifiedNamePart name="x"/>
          </exp:Identifier>
          <exp:RealLiteral>5.0</exp:RealLiteral>
        </exp:LogLeq>
      </exp:And>
    </exp:And>
  </fun:Condition>
  <fun:Message>
    "Variable x out of limit"
  </fun:Message>
</fun:Assertion>
```
3.1.5 Equations Module

In “FmiExtendedModelDescription.xsd” schema, the equation sections of models written in Modelica are represented under the “equ” namespace. The “equ” namespace includes the following complex types and elements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Type</td>
<td>AbstractEquation</td>
</tr>
<tr>
<td>Complex Type</td>
<td>BindingEquation</td>
</tr>
<tr>
<td>Complex Type</td>
<td>FunctionCallLeft</td>
</tr>
<tr>
<td>Complex Type</td>
<td>FunctionCallEquation</td>
</tr>
<tr>
<td>Element</td>
<td>DynamicEquations</td>
</tr>
<tr>
<td>Element</td>
<td>BindingEquations</td>
</tr>
<tr>
<td>Element</td>
<td>InitialEquations</td>
</tr>
</tbody>
</table>

Table 3.5: "equ" namespace complex types and elements
Complex type “BindingEquation” of equations module is defined as:

It is composed by two elements: "Parameter” defined as “QualifiedName” complex type which represents the left hand side of equation, and “BindingExp” which represents the left hand side of equation.

The “BindingEquations” element is defined as “BindingEquation” complex type as shown below in figure 3.20. It represents the set of all binding equations of models written in Modelica.

Example10. Representation of binding equations

Given binding equations in Modelica language (e.g., parameter Real a=3, b=2), the resulting XML representation valid according to the XML schema is:

```xml
<equ:BindingEquations>
  <equ:BindingEquation>
    <equ:Parameter>
      <exp:QualifiedNamePart name="a"/>
    </equ:Parameter>
    <equ:BindingExp>
      <exp:RealLiteral>3.0</exp:RealLiteral>
    </equ:BindingExp>
  </equ:BindingEquation>
</equ:BindingEquations>
```
Complex type “AbstractEquation” of equations module is defined as:

It represents equations in residual form i.e. exp1-exp2=0.

The “InitialEquations” element of equations module used to represent the set of initial equations defined in models written in Modelica. It accepts a list, possibly empty, of “Equation” elements defined as complex type “AbstractEquation”.

Example 11. Representation of initial equations
Given initial equations in Modelica language (e.g., Real x (start=0.1), y (start=0.1)), the resulting XML representation valid according to the XML schema is:

```xml
<equ:InitialEquations>
  <equ:Equation>
    <equ:Sub>
      <exp:Identifier>
        <exp:QualifiedNamePart name="x"/>
      </exp:Identifier>
      <exp:RealLiteral>0.1</exp:RealLiteral>
    </equ:Sub>
  </equ:Equation>
  <equ:Equation>
    <equ:Sub>
      <exp:Identifier>
        <exp:QualifiedNamePart name="y"/>
      </exp:Identifier>
      <exp:RealLiteral>0.1</exp:RealLiteral>
    </equ:Sub>
  </equ:Equation>
</equ:InitialEquations>
```

Complex type “FunctionCallEquation” of equations module is defined as:
Figure 3.23: "FunctionCallEquation" complex type definition

It represents function call equations with multiple outputs i.e. in the form of

\[(\text{output}1, \text{output}2, \text{output}N) = f(\text{input}1, \text{input}2, \ldots, \text{input}M)\]

where \text{output}1, \text{output}2\ldots\text{output}N are the left hand side of the equation that can be any combination of array, record constructors, or scalar variable identifier and \text{input}1,\text{input}2, \ldots, \text{input}M are the right hand side of the equation that can be any combination of scalar variable identifier or empty argument.

The “DynamicEquations” element of the equation module is defined as:

Figure 3.24: "DynamicEquations" element definition

It consists of a list of “Equation” elements defined as “AbstractEquation” complex type, that represent equations of a model written in Modelica which are in residual form, and
“FunctionCallEquation” elements that represent function call equations with multiple outputs.

**Example 12. Representation of dynamic equations**

Given dynamic equations in Modelica language (e.g., \( \text{der}(x) = y \)), the resulting XML representation valid according to the XML schema is:

```xml
<equ:DynamicEquations>
  <equ:Equation>
    <exp:Sub>
      <exp:Der>
        <exp:Identifier>
          <exp:QualifiedNamePart name="y"/>
        </exp:Identifier>
      </exp:Der>
      <exp:Identifier>
        <exp:QualifiedNamePart name="x"/>
      </exp:Identifier>
    </exp:Sub>
  </equ:Equation>
</equ:DynamicEquations>
```

### 3.1.6 Optimization Module

The optimization problems in “FmiExtendedModelDescription.xsd” schema are represented in the “opt” namespace. The “opt” namespace includes the following complex types and element:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Type</td>
<td>Constraint</td>
</tr>
<tr>
<td>Complex Type</td>
<td>TimeVariable</td>
</tr>
<tr>
<td>Element</td>
<td>Optimization</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
<td>Constraint complex type definition</td>
</tr>
<tr>
<td>TimeVariable</td>
<td>Time variable type definition</td>
</tr>
<tr>
<td>Optimization</td>
<td>Optimization problem representation</td>
</tr>
</tbody>
</table>

Table 3.6: “opt” namespace complex types and element

An overall optimization module in “FmiExtendedModelDescription.xsd” schema is defined in “optimization” element as shown below in figure 3.25.
Figure 3.25: An Overall Optimization module schema definition

Complex type “Constraint” of optimization module is defined as:

Figure 3.26: "Constraint" complex type definition
Complex type “TimeVariable” of optimization module is defined as:

![Figure 3.27: "TimeVariable" complex type definition](image)

The “Optimization” element consists of the following elements: “ObjectiveFunction”, ”IntegrandObjectiveFunction”, ”IntervalStartTime”, ”IntervalFinalTime”, and “Constraints”. The “ObjectiveFunction” element is composed by “TimedVariable” element which is represented by an expression. The “TimedVariable” element consists of two elements: “Identifier” which is defined as qualified name, and “Instant” that holds numerical value, see also figure 3.27 below.

![Figure 3.28: "TimedVariable" element definition](image)

**Example 12. Representation of constraints**

Given constraints in Optimica language (e.g., **constraint** \( u \leq 0.75 \)), the resulting XML representation valid according to the XML schema is:

```xml
<opt:Constraints>
  <opt:ConstraintLeq>
    <exp:Identifier>
      <exp:QualifiedNamePart name="u"/>
    </exp:Identifier>
    <exp:RealLiteral>0.75</exp:RealLiteral>
  </opt:ConstraintLeq>
</opt:Constraints>
```
Chapter 4

Implementation

This chapter describes the implementation of this thesis work. The first section presents the process of implantation in OMC. The second section explains how the XML code generator for OMC is implemented. Finally the third section explains how the template based Modelica Unparser for OMC is implemented.

4.1 Overview

First we write Modelica models (i.e. .mo files) and pass them to OMC. The source program passes through different transformation stages before reaching the code generator. Finally the code generator translates the internal representation of the source program to an XML code. An overview of the whole process can be summarized as shown in figure 4.1.

![Figure 4.1: The process of compiling Modelica model to XML code in OMC](image)

The flattened Modelica DAE from OMC Front-end is passed to the Back-end. Then, the Back-end sorts equations and performs optimizations and store in SimCode module for code generation using Susan template. Since this thesis work is mainly support for XML code generation and DAE Unparser, the most important part is the SimCode module.

The details of the data structure to generate different target code are not included in this report. We only explain the definition of SimCode, ModelInfo, and SimVars.
uniontype. Interested readers to know the full data structure representation are encouraged to read the source code from SimCode module.

SimCode is the root data structure for representing solved equation code as uniontype in MetaModelica definition. The uniontype definition of SimCode is shown below in Listing 4.1.

```meta
uniontype SimCode
  record SIMCODE
    ModelInfo modelInfo;
    list<DAE.Exp> literals;
    list<RecordDeclaration> recordDecls;
    list<String> externalFunctionIncludes;
    list<SimEqSystem> allEquations;
    list<list<SimEqSystem>> odeEquations;
    list<SimEqSystem> algebraicEquations;
    list<SimEqSystem> residualEquations;
    list<SimEqSystem> startValueEquations;
    list<SimEqSystem> parameterEquations;
    list<SimEqSystem> removedEquations;
    list<SimEqSystem> algorithmAndEquationAsserts;
    //list<DAE.Statement> algorithmAndEquationAsserts;
    list<DAE.Constraint> constraints;
    list<BackendDAE.ZeroCrossing> zeroCrossings;
    list<SimVar> zeroCrossingsNeedSave;
    list<SampleCondition> sampleConditions;
    list<SimEqSystem> sampleEquations;
    list<HelpVarInfo> helpVarInfo;
    list<SimWhenClause> whenClauses;
    list<DAE.ComponentRef> discreteModelVars;
    ExtObjInfo extObjInfo;
    MakefileParams makefileParams;
    DelayedExpression delayedExps;
    list<JacobianMatrix> jacobianMatrixes;
    Option<SimulationSettings> simulationSettingsOpt;
    String fileNamePrefix;
  end SIMCODE;
end SimCode;
```

Listing 4.1: SimCode Uniontype data structure

It also contains ModelInfo uniontype data structure that stores variable information and list of functions of the model. The uniontype definition of ModelInfo is shown below in listing 4.2.
The `ModelInfo` uniontype consists of two union types: `VarInfo`, which represents the number of different kind of variables in the model and `SimVars`, which represents the detail informations such as name, kind, min value, max value, causality etc., of the variable, see also listing 4.3.

```plaintext
uniontype ModelInfo
record MODELINFO
    Absyn.Path name;
    String directory;
    VarInfo varInfo;
    SimVars vars;
list<Function> functions;
list<String> labels;
end MODELINFO;
end ModelInfo;
```

Listing 4.2: ModelInfo Uniontype data structure

```plaintext
uniontype SimVars
record SIMVARS
    list<SimVar> stateVars;
    list<SimVar> derivativeVars;
    list<SimVar> algVars;
    list<SimVar> intAlgVars;
    list<SimVar> boolAlgVars;
    list<SimVar> inputVars;
    list<SimVar> outputVars;
    list<SimVar> aliasVars;
    list<SimVar> intAliasVars;
    list<SimVar> boolAliasVars;
    list<SimVar> paramVars;
    list<SimVar> intParamVars;
    list<SimVar> boolParamVars;
    list<SimVar> stringAlgVars;
    list<SimVar> stringParamVars;
    list<SimVar> stringAliasVars;
    list<SimVar> extObjVars;
    list<SimVar> jacobianVars; // all vars for the matrices A,B,C,D
    list<SimVar> constVars;
    list<SimVar> intConstVars;
    list<SimVar> boolConstVars;
    list<SimVar> stringConstVars;
end SIMVARS;
end SimVars;
```

Listing 4.3: SimVars Uniontype data structure

### 4.2 XML Code Generation in OpenModelica

An XML code generation in OMC using Susan template language starts from the root and traverses down to the branches of `SimCode` data structure. This is implemented by passing the necessary data structure as function arguments in Susan code. The overall process to generate an XML code is written in approximately 4,000 lines of codes.

Due to large amount of implementation code, a general overview of a top level template and main functions to generate XML code are described in the following sub-sections. However,
interested readers to know how the full implementation is done are encouraged to read the source code of CodegenXML.tpl file available in [12].

4.2.1 Result Output
The top level template of XML code generation starts with creation of text file which can be seen in listing 4.4.

```
template translateModel(SimCode simCode)
  "Generates root template for compiling a simulation of a Modelica model."
  ::= match simCode
  case SIMCODE(modelInfo = MODELINFO(__)) then
    let() = textFile(generateXml(simCode), '<%dotPathXml(modelInfo.name)%>.xml')
    "" //always returns an empty result since generated texts are written to files directly
  end translateModel;
```

Listing 4.4 Top level template for XML code generation

The top level template does not return anything rather it uses a special function `textFile()` to output all the results of main function to an .xml file.

4.2.2 The main function
The main template consists of a set of templates to generate an XML file for variables, variable declarations, equations, algorithms, functions, records and optimization section of a model. The main template “generateXml()” is defined as shown below in listing 4.5.

```
template generateXml(SimCode simCode)
  "Generates XML code for."
  ::= match simCode
  case SIMCODE(modelInfo = MODELINFO(__)) then
    let guid = getUUIDStr()
    <<
      <?xml version="1.0" encoding="UTF-8"?>
      <OpenModelicaModelDescription>
        xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
        xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
        xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
        xmlns:opt="https://svn.jmodelica.org/trunk/XML/daeOptimization.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        <%modelDescriptionXml(simCode, guid)%>
        <%vendorAnnotationsXml(simCode)%>
        <%defaultExperiment(simulationSettingsOpt)%>
        <%modelVariablesXml(modelInfo)%>
        <%bindingEquationsXml(modelInfo)%>
        <%equationsXml(allEquations, whenClauses, helpVarInfo)%>
        <%algorithmicEquationsXml(allEquations)%>
      </OpenModelicaModelDescription>
    >>
  end generateXml;
```

...
4.2.3 Qualified Names XML Code generation

The XML code generation template corresponding to the qualified name parts of the XML schema (see section 3.1.1) is defined below in Listing 4.6.

```plaintext
template qualifiedNamePartXml(ComponentRef cr)
  "Generates XML code of the Qualified name of a variable."
  ::= match cr
case CREF_IDENT(__) then
  let arrayTest = arraysubscriptsStrXml(subscriptLst)
  if arrayTest then
    <<
      <exp:QualifiedNamePart name="<%ident%>">
        %arraysubscriptsStrXml(subscriptLst)%
      </exp:QualifiedNamePart>
    >>
  else
    <<
      <exp:QualifiedNamePart name="<%ident%/>">
        %qualifiedNamePartXml(componentRef)%
      </exp:QualifiedNamePart>
    >>
  case CREF_QUAL(ident = "$DER") then '<%qualifiedNamePartXml(componentRef)%>'
case CREF_QUAL(__) then
  let arrayTest = arraysubscriptsStrXml(subscriptLst)
  if arrayTest then
    <<
      <exp:QualifiedNamePart name="<%ident%>">
        %arraysubscriptsStrXml(subscriptLst)%
        %qualifiedNamePartXml(componentRef)%
      </exp:QualifiedNamePart>
    >>
  else
    <<
      <exp:QualifiedNamePart name="<%ident%/>">
        %qualifiedNamePartXml(componentRef)%
      </exp:QualifiedNamePart>
    >>
  else "CREF_NOT_IDENT_OR_QUAL"
end qualifiedNamePartXml;
```

Listing 4.6: Parts of code for qualified name of variables XML code generation

The code generation starts from the “qualifiedNamePartXml()” template for every qualified name part which generates XML code of the qualified name of a variable, which in
turn calls the “arraySubscriptsStrXml()” template if the qualified name part has subscript. The “arraySubscriptsStrXml()” template is shown below in listing 4.7.

```cpp
template arraySubscriptsStrXml(list<Subscript> subscripts)
"Generates XML code for subscript part of the name."
::=
  if subscripts then
    <<
      <exp:ArraySubscripts>
        <%subscripts | s => arraySubscriptStrXml(s) ; separator="\n"%>
      </exp:ArraySubscripts>
    >>
  else
    <<<
end arraySubscriptsStrXml;

template arraySubscriptStrXml(Subscript subscript)
"Generates a single subscript XML code. Only works for constant integer indicies."
::=
  match subscript
  case INDEX(exp=ICONST(integer=i)) then
    <<
      <exp:IndexExpression>
        <exp:IntegerLiteral><%i%></exp:IntegerLiteral>
      </exp:IndexExpression>
    >>
  case SLICE(exp=ICONST(integer=i)) then
    <<
      <exp:IndexExpression>
        <exp:IntegerLiteral><%i%></exp:IntegerLiteral>
      </exp:IndexExpression>
    >>
  case WHOLEDIM(__) then "WHOLEDIM"
  else "UNKNOWN_SUBSCRIPT"
end arraySubscriptStrXml;
```

Listing 4.7: Parts of code for array subscripts of qualified name XML code generation

It starts the XML code generation for array subscripts by generating the opening tag “<exp:ArraySubscripts>” and calls “arraySubscriptStrXml()” template for every subscript and finally generates the closing tag “</exp:ArraySubscripts>”. The “arraySubscriptStrXml()” template generates XML code for single subscript of qualified name part.

4.2.5 Equations XML Code generation

Based on the equation module of the XML schema defined in section 3.15, the module consists of three parts: Initial equations, binding equations, and dynamic equations. The implementation of the XML code generation for equation definitions of a model is also correspondent to those parts. The “initialEquationsXml()” template function(see Listing 4.6 below) generates the opening tag ”<equ:InitialEquations>” and calls the
“initialEquationXml()” template function for each type of variables. Finally it generates the closing tag”</equ:InitialEquations>”.

```
template initialEquationsXml(ModelInfo modelInfo)

"Function for Initial Equations."
::=
match modelInfo
case MODELINFO(varInfo=VARINFO(numStateVars=numStateVars),vars=SIMVARS(____)) then
  <<
  <equ:InitialEquations>
    <%vars.stateVars | var => initialEquationXml(var) ;separator="\n">%>var
    <%vars.derivativeVars | var => initialEquationXml(var) ;separator="\n">%>var
    <%vars.intAlgVars | var => initialEquationXml(var) ;separator="\n">%>var
    <%vars.boolAlgVars | var => initialEquationXml(var) ;separator="\n">%>var
    <%vars.stringAlgVars | var => initialEquationXml(var) ;separator="\n">%>var
  </equ:InitialEquations>
>>
end initialEquationsXml;
```

Listing 4.8: Root template for Initial equations XML code generation

The “initialEquationXml()” template receives the “SimVars” as an argument and matches its initial value field. If its value is “true”, it generates the XML code of the variable defined as initial equation and returns the result to “initialEquationsXml()”.

The binding equation and dynamic equations portions of the XML schema are also generated in the same manner as XML code generation for initial equations. Interested readers to know how the full implementation of the equation module is done are encouraged to read the source code of CodegenXML .tpl file available in [12].

4.2.6 Functions XML Code generation

The root template for XML code generation of the user defined functions corresponding to the XML schema (See section 3.1.3) is defined as below in listing 4.10. The “functions” field in the “MODELINFO” record contains a list of Modelica functions declared as “list<Function>”.

```
template functionsXml(list<Function> functions)

"Generates the body for a set of functions."
::=
<<
  <fun:FunctionList>
    <%functions | fn => functionXml(fn) ;separator="\n">%>fn
  </fun:FunctionList>
>>
end functionsXml;
```

Listing 4.9: Root template for functions XML code generation

The root template “functionsXml( )” starts the XML code generation by generating the opening tag “<fun:FunctionList>” and calls “functionXml()” template for every
function definitions found in the “list<Function>”. Finally it generates the closing tag “</fun:FunctionList>”. Interested readers to know how “functionXml()” template is done are encouraged to read the source code of CodegenXML.tpl file available in [12].

4.2.7 Algorithms XML Code generation
The root template for XML code generation of algorithmic equations corresponding to the XML schema (See section 3.1.4) is defined as below in listing 4.10. The “allEquations” field in the “SIMCODE” record contains a list of Modelica algorithmic equations declared as “list<SimEqSystem>”. The body of the algorithm section can be a list of statements such as an assignment, a while loop, for loop etc.

```
template algorithmicEquationsXml(list<SimEqSystem> allEquations)
    "Generates XML for an equation that is an algorithm."
::=
let &varDecls = buffer "" /*BUFD*/
let algs = (allEquations |> eq =>
    equationAlgorithmXml(eq, contextSimulationDiscrete, &varDecls /*BUFD*/)
;separator="\n")
<<
<fun:Algorithm>
  <%algs%>
</fun:Algorithm>
>>
end algorithmicEquationsXml;
```

Listing 4.10: Root template for Algorithms XML code generation

For every algorithmic equation definitions found in the “list<SimEqSystem>”, the root template calls “equationAlgorithmXml()” template which generates XML code for the body part of each algorithm. The output of “equationAlgorithmXml()” template is assigned to a variable “algs”.

The root template “algorithmicEquationsXml( )” starts the XML code generation by generating the opening tag “<fun:Algorithm>” and the body of the algorithm which is the output of “algs”. Finally it generates the closing tag “</fun:Algorithm>”.

Interested readers to know how “equationAlgorithmXml()” template is done are encouraged to read the source code of CodegenXML.tpl file available in [12].

4.2.8 Records XML Code generation
The root template for XML code generation of the records definition corresponding to the XML schema defined in section 3.1.3 is defined below in listing 4.11.

```
template recordsXml(list<RecordDeclaration> recordDecls)
    "Generates XML code for all records."
::=
<<
```

56
The root template “recordsXml ( )” starts the XML code generation by generating the opening tag “<fun:RecordsList>” and calls “recordDeclarationXml ( )” template for every record definitions found in the “list<RecordDeclaration>”. Finally it generates the closing tag “</fun:RecordsList>”. Interested readers to know how “recordDeclarationXml ( )” template is done are encouraged to read the source code of CodegenXML.tpl file available in [12].

### 4.3 Template Based DAE Unparser

#### 4.3.1 The root Data Structure

The details of the data structure used for DAE unparser are not included in this report. We only explain the root data structure definition which is compWithSplitElements. Interested readers to know the full data structure representation are encouraged to read the source code of DAEDumpTV.

compWithSplitElements and functionList are the root data structure. They are defined as uniontype in MetaModelica definition. The uniontype definition of compWithSplitElements and functionList can be seen in Listing 4.12 and 4.14 respectively.

```plaintext
uniontype compWithSplitElements
record COMP_WITH_SPLIT
    String name;
    splitElements spltElems;
    Option<SCode.Comment> comment;
end COMP_WITH_SPLIT;
end compWithSplitElements;
```

**Listing 4.12: compWithSplitElements Uniontype data structure**

compWithSplitElements structure represents name of the component, an optional comment for the component defined as Option<SCode.Comment>, and uniontype splitElements structure that stores variables, initial equations, initial algorithms, equations, algorithms. The uniontype splitElements structure can be seen in listing 4.13 below.

```plaintext
uniontype splitElements
record SPLIT_ELEMENTS
    list<DAE.Element> v;
    list<DAE.Element> ie;
```

**Listing 4.13: splitElements Uniontype data structure**
The `uniontype` functionList structure represents list of sorted functions defined as

```plaintext
uniontype functionList
    record FUNCTION_LIST
        list<DAE.Function> funcs;
    end FUNCTION_LIST;
end functionList;
```

### 4.3.2 Template Implementation of DAE Unparser

In this section we present the root template intended for transforming OpenModelica AST into text. However, interested readers to know how the full implementation is done are encouraged to read the source code of DAEDumpTpl.tpl file available in [12].

The `dumpDAE` template is the top level template which can be seen in listing 4.15 below. It takes a set of DAE elements (`list<DAEDump.compWithSplitElements>`) and list of functions (`DAEDump.functionList`).

```plaintext
template dumpDAE(list<DAEDump.compWithSplitElements> fixedDaeList, DAEDump.functionList  funLists)
::=
let comp_str =(fixedDaeList |> dae => dumpComp(dae) ;separator="\n")
let fun_str = match funLists case FUNCTION_LIST(__) then dumpFunctions(funcs)
if fun_str then
  <<
  <%fun_str%>
  <%\n\n%>
  <%comp_str%>
>>
else
  <<
  <%comp_str%>
>>
end dumpDAE;
```

### Listing 4.15: dumpDAE template-top level template for DAE Unparser

The `dumpDAE` calls two templates: `dumpComp` template for each component to output the actual definitions of variables, initial equations, initial algorithms, and equations, algorithms, and, and `dumpFunctions` template to output list of functions. The `dumpComp` and `dumpFunctions` template are shown below in listing 4.16 and 4.17 respectively.

```plaintext
template dumpComp(DAEDump.compWithSplitElements fixedDae)
::=
```
match fixedDae case COMP_WITH_SPLIT(__) then
  let cmt_str = dumpCommentOpt(comment)
  <<
  class <\%name%><\%cmt_str%>
  <\%dumpCompStream(splitElems)%>
  end <\%name%>;<\%\n%>
>>
end dumpComp;

template dumpCompStream(DAEDump.splitElements elems)
::= 
  match elems
  case SPLIT_ELEMENTS(__) then
    let var_str = dumpVars(v)
    let ieq_str = dumpInitialEquations(ie, "initial equation")
    let ial_str = dumpInitialAlgorithms(ia, "initial algorithm")
    let eq_str = dumpEquations(e, "equation")
    let al_str = dumpAlgorithms(a, "algorithm")
    <<
    <\%var_str%>
    <\%ieq_str%>
    <\%ial_str%>
    <\%eq_str%>
    <\%al_str%>
  >>
end dumpCompStream;

Listing 4.16: dumpComp and dumpCompStream templates to unparse components and elements

The dumpComp takes DAEDump.compWithSplitElements and returns the component name and comment. It also calls dumpCompStream template which in turn calls other templates to output the different parts such as variables, initial equations, initial algorithms, equations, and algorithms.

template dumpFunctions(list<DAE.Function> funcs)
::= 
  (funcs |> func => dumpFunction(func);separator="\n\n")
end dumpFunctions;

template dumpFunction(DAE.Function function)
::= 
  match function
    case FUNCTION(functions=FUNCTION_EXT(externalDecl =
      EXTERNALDECL(language="builtin"));__) then ''
    case FUNCTION(__) then
      let cmt_str = dumpCommentOpt(comment)
      <<
      function <\%AbsynDumpTpl.dumpPathNoQual(path)%><\%cmt_str%>
      <\%dumpFunctionDefinitions(functions)%>
      end <\%AbsynDumpTpl.dumpPathNoQual(path)%>;
      >>
    case RECORD_CONSTRUCTOR(__) then
      <<
      function <\%AbsynDumpTpl.dumpPathNoQual(path)%> "Automatically generated
The dumpFunctions template takes list of functions (list<DAE.Function) and calls dumpFunction template for each function found in the list. The dumpFunction template matches DAE.Function against FUNCTION and RECORD_CONSTRUCTOR record types.

- If it matches with the FUNCTION record type, it will unpars the function name and calls dumpFunctionDefinitions template to unpars the actual definition of the function.

- If it matches with the RECORD_CONSTRUCTOR record type, it will unpars the name of the record constructor and calls dumpRecordInputVarStr template to unpars the actual declaration of the record.
Chapter 5

Test Implementation

This chapter presents different test cases conducted to test the implementation of the thesis work.

5.1 Test Models for XML Code generation

To be able to test the implementation of XML code generation module it is necessary to choose some models written in Modelica and Optimica. Three models have been taken into account for this purpose. They were chosen because of their use of different Modelica language constructs. The models that were chosen and their corresponding XML exported from OMC are described in the following sub-sections.

5.1.1 RecordVariability.mo

This model is taken from OpenModelica test suite associated with OMC source code. The Modelica code for this model can be seen in listing 5.1.

```
record abcRec
    Integer a;
    parameter Integer b = 2;
    constant Integer c = 3;
end abcRec;

model recordVariability
    constant Integer p = 13;
    constant abcRec x = abcRec(1);
    constant abcRec y = abcRec(4,p*2);
    parameter abcRec z = abcRec(2,p);
end recordVariability;
```

Listing 5.1: The Modelica code for the recordVariability model

The row Modelica model is then flattened into the following code before XML code is generated.
function abcRec "Automatically generated record constructor for abcRec"
    input Integer a;
    input Integer b = 2;
    constant Integer c = 3;
    output abcRec res;
end abcRec;

class recordVariability
    constant Integer p = 13;
    constant Integer x.a = 1;
    constant Integer x.b = 2;
    constant Integer x.c = 3;
    Integer z.a = 2;
    parameter Integer z.b = 13;
    constant Integer z.c = 3;
    parameter Integer y.a = 4;
    parameter Integer y.b = 26;
    constant Integer y.c = 3;
end recordVariability;

Listing 5.2: The flattened Modelica code for the recordVariability model

Finally an XML code representing the model is exported from OpenModelica. The full XML code result is given in Appendix A.

5.1.2 Cross.mo
This model is also taken from OpenModelica test suite associated with OMC source code. The Modelica code for this model can be seen in listing 5.3.

model Cross
    function myCrossReal
        input Real[3] x;
        input Real[3] y;
        output Real[3] z;
    algorithm
        z := cross(x,y);
    end myCrossReal;

    function myCrossInt
Listing 5.3: The Modelica code for the Cross model

The row Modelica model is then flattened into the following code before XML code is generated.

```modelica
function Cross.myCrossInt "Shouldn't be elaborated to equations like cross sometimes is"
    input Integer[3] x;
    input Integer[3] y;
    output Real[3] z;
    protected
        Integer[3] ztmp;
    algorithm
        ztmp := cross(x,y);
        z := ztmp;
    end myCrossInt;

Real x[3] = {1,5,3};
Real y1[3] = {2,10,6};
Real y2[3] = {5,3,1};
Real[3] z;
Integer xi[3] = {1,5,3};
Integer yi1[3] = {2,10,6};
Integer yi2[3] = {5,3,1};
discrete Real[3] zi;
equation
    z = myCrossReal(x, if time > 0.1 then y2 else y1);
    zi = myCrossInt(xi, if time > 0.1 then yi2 else yi1);
end Cross;
```
function Cross.myCrossReal "Shouldn't be elaborated to equations like cross sometimes is"
  input Real[3] x;
  input Real[3] y;
  output Real[3] z;
algorithm
end Cross.myCrossReal;

class Cross
  Real x[1] = 1.0;
  Real x[2] = 5.0;
  Real x[3] = 3.0;
  Real y1[1] = 2.0;
  Real y1[2] = 10.0;
  Real y1[3] = 6.0;
  Real y2[1] = 5.0;
  Real y2[2] = 3.0;
  Real y2[3] = 1.0;
  Real z[1];
  Real z[2];
  Real z[3];
  Integer xi[1] = 1;
  Integer xi[2] = 5;
  Integer xi[3] = 3;
  Integer yi1[1] = 2;
  Integer yi1[2] = 10;
  Integer yi1[3] = 6;
  Integer yi2[1] = 5;
  Integer yi2[2] = 3;
  Integer yi2[3] = 1;
  discrete Real zi[1];
  discrete Real zi[2];
  discrete Real zi[3];
equation
  z = Cross.myCrossReal({x[1], x[2], x[3]}, if time > 0.1 then {y2[1], y2[2], y2[3]} else {y1[1], y1[2], y1[3]});
Finally an XML code representing the model is exported from OpenModelica. The full XML code result is given in Appendix B.

5.1.3 VDP_Opt.mo
The last model, VDP_Opt, is taken from [4]. It models an optimal control problem on Van der Pol oscillator model written in Optimica code. The Optimica code for this model can be seen in listing 5.5.

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

// Parameters
parameter Real p1 = 1;
parameter Real p2 = 1;
parameter Real p3 = 2;
// The states
Real x1(start = 0);
Real x2(start = 1);
// The control signal
input Real u;
Real cost(start=0);
equation
  der(x1) = (1 - x2^2) * x1 - x2 + u;
  der(x2) = p1 * x1;
  der(cost)=exp(p3*1/*time*/) * (x1^2 + x2^2 + u^2);
constraint u<=0.75;
end VDP_Opt;
```

The model is then flattened into the following code before XML code is generated.
Optimization VDP_Opt (objective = cost(finalTime),
   startTime = 0, finalTime = 20)

parameter Real p1 = 1;
parameter Real p2 = 1;
parameter Real p3 = 2;
Real x1(start = 0, fixed=true);
Real x2(start = 1, fixed=true);
input Real u;
Real cost(start=0, fixed=true);
Real der(x1);
Real der(x2);
Real der(cost);
equation
der(x1) = (1 - (x2 ^ 2)) * (x1) - (x2) + u;
der(x2) = (p1) * (x1);
der(cost) = (exp(p3)) * (x1^2 + x2^2 + u^2);
constraint u<=0.75;
end VDP_Opt;

Listing 5.6: The flattened Modelica code for the VDP_Opt model

Finally an XML code representing the model is exported from OpenModelica. The full XML code result is given in Appendix C.

5.2 Test Models for DAE Unparser
To be able to evaluate the implementation of DAE Unparser, we run the test suite consists of 630 Modelica files (.mo) in the directory "/testsuite/mofiles" of OMC source code and all test cases passed.
Chapter 6

Conclusion and Future Work

This chapter concludes the work conducted in this thesis and discusses the possible directions for future work.

6.1 Conclusion
In this thesis, a new template based XML code generation module has been implemented for OMC that can successfully translate flattened Modelica model to an XML format. We have made a total of 100 example test cases and the test implementation of OMC has shown the possibility to use the XML representation to export Modelica models according to an XML standard defined in OPENPROD EU project. We have also implemented a new template based Modelica Unparser for OpenModelica AST into text, which is more concise and readable code comparing to the current OpenModelica Unparser written in MetaModelica.

6.2 Future Work
The current version of OMC has no support for dynamic optimization of optimal control problems, apart from compilation and simulation of models written in Modelica and Optimica language construct. In order to support dynamic optimization using OMC, as a future work, needs to integrate it with an open source software CasADi based on an XML standard for exchange of DAE models defined in [3][4]. The OMC supports export of models in this format which is part of the thesis work, whereas CasADi supports import of models expressed in this format.

Another task for future work is to implement a Click-on button in OMEdit which makes exporting models to an XML format from OpenModelica straightforward for users.
Appendix

A XML code Results

In the following section an XML code exported from OpenModelica will be presented.

A.1 RecordVariability.mo

<?xml version="1.0" encoding="UTF-8"?>
<OpenModelicaModelDescription>
xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
xmlns:opt="https://svn.jmodelica.org/trunk/XML/daeOptimization.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:fmiVersion="1.0"
xmlns:guid="{06a75b9c-3b98-4f56-ba79-017cdec8971d}"
generationDateAndTime="2012-05-18T17:07:45"
variableNamingConvention="Structured"
numberOfContinuousStates="1"
numberOfEventIndicators="0"
<VendorAnnotations>
<Tool name="OpenModelica Compiler 1.8.1+ (r11925)">
</Tool>
</VendorAnnotations>
<defaultExperiment startTime="0.0" stopTime="1.0" tolerance="1e-006"/>
<ModelVariables>
<ScalarVariable name="z.a" valueReference="2" variability="discrete"
causality="internal" alias="noAlias">
<Integer start="2" fixed="false"/>
<QualifiedName>
<exp:QualifiedNamePart name="z"/>
<exp:QualifiedNamePart name="a"/>
</QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="z.b" valueReference="3" variability="parameter"
causality="internal" alias="noAlias">
<Integer start="13" fixed="true"/>
<QualifiedName>
<exp:QualifiedNamePart name="z"/>
<exp:QualifiedNamePart name="b"/>
</QualifiedName>
<VariableCategory>Parameter</VariableCategory>
</ScalarVariable>
<ScalarVariable name="y.a" valueReference="4" variability="parameter"
causality="internal" alias="noAlias">
<Integer start="4" fixed="true"/>
<QualifiedName>
</ScalarVariable>
</ModelVariables>
<QualifiedName>
    <exp:QualifiedNamePart name="z"/>
    <exp:QualifiedNamePart name="c"/>
</QualifiedName>
<VariableCategory>Constant</VariableCategory>
<ScalarVariable>
    <ScalarVariable name="y.c" valueReference="11" variability="constant"
        causality="internal" alias="noAlias">
        <Integer />
    </ScalarVariable>
</ModelVariables>
<equ:BindingEquations>
    <equ:BindingEquation>
        <equ:Parameter>
            <exp:QualifiedNamePart name="z"/>
            <exp:QualifiedNamePart name="b"/>
        </equ:Parameter>
        <equ:BindingExp>
            <exp:IntegerLiteral>13</exp:IntegerLiteral>
        </equ:BindingExp>
    </equ:BindingEquation>
    <equ:BindingEquation>
        <equ:Parameter>
            <exp:QualifiedNamePart name="y"/>
            <exp:QualifiedNamePart name="a"/>
        </equ:Parameter>
        <equ:BindingExp>
            <exp:IntegerLiteral>4</exp:IntegerLiteral>
        </equ:BindingExp>
    </equ:BindingEquation>
    <equ:BindingEquation>
        <equ:Parameter>
            <exp:QualifiedNamePart name="y"/>
            <exp:QualifiedNamePart name="b"/>
        </equ:Parameter>
        <equ:BindingExp>
            <exp:IntegerLiteral>26</exp:IntegerLiteral>
        </equ:BindingExp>
    </equ:BindingEquation>
</equ:BindingEquations>
<fun:Algorithm>
</fun:Algorithm>
<equ:InitialEquations>
    <equ:Equation>
        <equ:Sub>
            <exp:Identifier>
                <exp:QualifiedNamePart name="z"/>
            </exp:Identifier>
        </equ:Sub>
    </equ:Equation>
</equ:InitialEquations>
<exp:QualifiedNamePart name="a"/>
</exp:Identifier>
<exp:IntegerLiteral>2</exp:IntegerLiteral>
</equ:Sub>
</equ:Equation>
</equ:InitialEquations>

<fun:RecordsList>
<fun:Record>
<fun:Name>
<exp:QualifiedNamePart name='abcRec'/>
</fun:Name>
<fun:Field type="integer">
<fun:Name>
<exp:QualifiedNamePart name="a"/>
</fun:Name>
</fun:Filed>
<fun:Field type="integer">
<fun:Name>
<exp:QualifiedNamePart name="b"/>
</fun:Name>
</fun:Filed>
<fun:Field type="integer">
<fun:Name>
<exp:QualifiedNamePart name="c"/>
</fun:Name>
</fun:Filed>
</fun:Record>
</fun:RecordsList>

A.2 Cross.mo

<?xml version="1.0" encoding="UTF-8"?&gt;
<OpenModelicaModelDescription>
xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
xmlns:opt="https://svn.jmodelica.org/trunk/XML/daeOptimization.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
fmiVersion="1.0"
modelName="Cross"
modelIdentifier="Cross"
guid="{098c524-750b-41ee-bbc-2b733a4cd406}"
generationDateAndTime="2012-05-18T17:15:32"
variableNamingConvention="Structured"
numberOfContinuousStates="1"
numberOfEventIndicators="1"
<VendorAnnotations>
<Tool name="OpenModelica Compiler 1.8.1+ (r11925)"/>
</VendorAnnotations>
<defaultExperiment startTime="0.0" stopTime="1.0" tolerance="1e-006"/>
<ModelVariables>
  <ScalarVariable name="z[1]" valueReference="2" variability="continuous"
    causality="internal" alias="noAlias">
    <Real />
  </ScalarVariable>
  <QualifiedName>
    <exp:QualifiedNamePart name="z">
      <exp:ArraySubscripts>
        <exp:IndexExpression>
          <exp:IntegerLiteral>1</exp:IntegerLiteral>
        </exp:IndexExpression>
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
  </QualifiedName>
  <VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>

< ScalarVariable name="z[2]" valueReference="3" variability="continuous"
    causality="internal" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="z">
      <exp:ArraySubscripts>
        <exp:IndexExpression>
          <exp:IntegerLiteral>2</exp:IntegerLiteral>
        </exp:IndexExpression>
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
  </QualifiedName>
  <VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>

< ScalarVariable name="z[3]" valueReference="4" variability="continuous"
    causality="internal" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="z">
      <exp:ArraySubscripts>
        <exp:IndexExpression>
          <exp:IntegerLiteral>3</exp:IntegerLiteral>
        </exp:IndexExpression>
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
  </QualifiedName>
  <VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>

< ScalarVariable name="zi[1]" valueReference="5" variability="discrete"
    causality="internal" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="zi">
      <exp:ArraySubscripts>
        <exp:IndexExpression>
          <exp:IntegerLiteral>1</exp:IntegerLiteral>
        </exp:IndexExpression>
      </exp:ArraySubscripts>
    </exp:QualifiedNamePart>
  </QualifiedName>
  <VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="zi[2]" valueReference="6" variability="discrete" causality="internal" alias="noAlias">
  <Real />
</ScalarVariable>

<ScalarVariable name="zi[3]" valueReference="7" variability="discrete" causality="internal" alias="noAlias">
  <Real />
</ScalarVariable>

<ScalarVariable name="y2[1]" valueReference="8" variability="continuous" causality="internal" alias="noAlias">
  <Real start="5.0" fixed="false" />
</ScalarVariable>

<ScalarVariable name="y2[2]" valueReference="9" variability="continuous" causality="internal" alias="noAlias">
  <Real start="3.0" fixed="false" />
</ScalarVariable>
<exp:ArraySubscripts>
  <exp:IndexExpression>
    <exp:IntegerLiteral>2</exp:IntegerLiteral>
  </exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="y2[3]" valueReference="10" variability="continuous"
  causality="internal" alias="noAlias">
  <Real start="1.0" fixed="false" />
</ScalarVariable>
<exp:QualifiedNamePart name="y2">
  <exp:ArraySubscripts>
    <exp:IndexExpression>
      <exp:IntegerLiteral>3</exp:IntegerLiteral>
    </exp:IndexExpression>
  </exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="y1[1]" valueReference="11" variability="continuous"
  causality="internal" alias="noAlias">
  <Real start="2.0" fixed="false" />
</ScalarVariable>
<exp:QualifiedNamePart name="y1">
  <exp:ArraySubscripts>
    <exp:IndexExpression>
      <exp:IntegerLiteral>1</exp:IntegerLiteral>
    </exp:IndexExpression>
  </exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="y1[2]" valueReference="12" variability="continuous"
  causality="internal" alias="noAlias">
  <Real start="10.0" fixed="false" />
</ScalarVariable>
<exp:QualifiedNamePart name="y1">
  <exp:ArraySubscripts>
    <exp:IndexExpression>
      <exp:IntegerLiteral>2</exp:IntegerLiteral>
    </exp:IndexExpression>
  </exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="y1[3]" valueReference="13" variability="continuous"
<Real start="6.0" fixed="false" />  
<QualifiedName>  
  <exp:QualifiedNamePart name="y1">  
    <exp:ArraySubscripts>  
      <exp:IndexExpression>  
        <exp:IntegerLiteral>3</exp:IntegerLiteral>  
      </exp:IndexExpression>  
    </exp:ArraySubscripts>  
  </exp:QualifiedNamePart>  
</QualifiedName>  
<VariableCategory>Algebraic</VariableCategory>  
</ScalarVariable>  
<ScalarVariable name="x[1]" valueReference="14" variability="continuous" causality="internal" alias="noAlias">  
  <Real start="1.0" fixed="false" />  
  <QualifiedName>  
    <exp:QualifiedNamePart name="x">  
      <exp:ArraySubscripts>  
        <exp:IndexExpression>  
          <exp:IntegerLiteral>1</exp:IntegerLiteral>  
        </exp:IndexExpression>  
      </exp:ArraySubscripts>  
    </exp:QualifiedNamePart>  
  </QualifiedName>  
  <VariableCategory>Algebraic</VariableCategory>  
</ScalarVariable>  
<ScalarVariable name="x[2]" valueReference="15" variability="continuous" causality="internal" alias="noAlias">  
  <Real start="5.0" fixed="false" />  
  <QualifiedName>  
    <exp:QualifiedNamePart name="x">  
      <exp:ArraySubscripts>  
        <exp:IndexExpression>  
          <exp:IntegerLiteral>2</exp:IntegerLiteral>  
        </exp:IndexExpression>  
      </exp:ArraySubscripts>  
    </exp:QualifiedNamePart>  
  </QualifiedName>  
  <VariableCategory>Algebraic</VariableCategory>  
</ScalarVariable>  
<ScalarVariable name="x[3]" valueReference="16" variability="continuous" causality="internal" alias="noAlias">  
  <Real start="3.0" fixed="false" />  
  <QualifiedName>  
    <exp:QualifiedNamePart name="x">  
      <exp:ArraySubscripts>  
        <exp:IndexExpression>  
          <exp:IntegerLiteral>3</exp:IntegerLiteral>  
        </exp:IndexExpression>  
      </exp:ArraySubscripts>  
    </exp:QualifiedNamePart>  
  </QualifiedName>  
</ScalarVariable>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</QualifiedName>
</VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="yi1[2]" valueReference="21" variability="discrete"
causality="internal" alias="noAlias">
<Integer start="10" fixed="false"/>
<QualifiedName>
<exp:QualifiedNamePart name="yi1">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>2</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="yi1[3]" valueReference="22" variability="discrete"
causality="internal" alias="noAlias">
<Integer start="6" fixed="false"/>
<QualifiedName>
<exp:QualifiedNamePart name="yi1">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>3</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="xi[1]" valueReference="23" variability="discrete"
causality="internal" alias="noAlias">
<Integer start="1" fixed="false"/>
<QualifiedName>
<exp:QualifiedNamePart name="xi">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="xi[2]" valueReference="24" variability="discrete"
causality="internal" alias="noAlias">
<Integer start="5" fixed="false"/>
<QualifiedName>
  <exp:QualifiedNamePart name="xi">
    <exp:ArraySubscripts>
      <exp:IndexExpression>
        <exp:IntegerLiteral>2</exp:IntegerLiteral>
      </exp:IndexExpression>
    </exp:ArraySubscripts>
  </exp:QualifiedNamePart>
</QualifiedName>
<VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="xi[3]" valueReference="25" variability="discrete" causality="Internal" alias="noAlias">
  <Integer start="3" fixed="false"/>
</ScalarVariable>
</ModelVariables>
<equ:BindingEquations>
  <equ:DynamicEquations>
    <equ:Equation>
      <exp:Sub>
        <exp:Identifier>
          <exp:QualifiedNamePart name="zi"/>
        </exp:Identifier>
        <exp:FunctionCall>
          <exp:Name>
            <exp:QualifiedNamePart name="Cross"/>
            <exp:QualifiedNamePart name="myCrossInt"/>
          </exp:Name>
          <exp:Arguments>
            tmp2
          </exp:Arguments>
        </exp:FunctionCall>
      </exp:Sub>
    </equ:Equation>
    <equ:Equation>
      <exp:Sub>
        <exp:Identifier>
          <exp:QualifiedNamePart name="z"/>
        </exp:Identifier>
        <exp:FunctionCall>
          <exp:Name>
            ...
          </exp:Name>
        </exp:FunctionCall>
      </exp:Sub>
    </equ:Equation>
  </equ:DynamicEquations>
</equ:BindingEquations>
<exp:QualifiedNamePart name="Cross"/>
<exp:QualifiedNamePart name="myCrossReal"/>
</exp:Name>
<exp:Arguments>
tmp6
</exp:Arguments>
</exp:FunctionCall>
</exp:Sub>
</equ:Equation>
</equ:DynamicEquations>
</fun:Algorithm>
</fun:Algorithm>
<equ:InitialEquations>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y2">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
<exp:IntegerLiteral>1</exp:IntegerExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:Identifier>
<exp:RealLiteral>5.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y2">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>2</exp:IntegerLiteral>
<exp:IntegerLiteral>2</exp:IntegerExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:Identifier>
<exp:RealLiteral>3.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y2">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>3</exp:IntegerLiteral>
<exp:IntegerLiteral>3</exp:IntegerExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
</exp:Identifier>
</equ:Sub>
</equ:Equation>
</equ:InitialEquations>
<exp:RealLiteral>1.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y1">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
<exp:RealLiteral>2.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y1">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>2</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
<exp:RealLiteral>10.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="y1">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>3</exp:IntegerLiteral>
<exp:IndexExpression>
</exp:ArraySubscripts>
</exp:QualifiedNamePart>
<exp:RealLiteral>6.0</exp:RealLiteral>
</equ:Sub>
</equ:Equation>
<equ:Equation>
<equ:Sub>
<exp:Identifier>
<exp:QualifiedNamePart name="x">
<exp:ArraySubscripts>
<exp:IndexExpression>
<exp:IntegerLiteral>1</exp:IntegerLiteral>
</exp:IndexExpression>
</exp:Sub>
</equ:Equation>
\[ \begin{align*}
    x_{2} &= 1.0, \\
    x_{3} &= 5.0, \\
    y_{12} &= 5.0, \\
    y_{21} &= 3.0. 
\end{align*} \]
\[ \text{yi2}[2][3] = 3 \]
\[ \text{yi2}[3][1] = 1 \]
\[ \text{yi1}[1][2] = 2 \]
\[ \text{yi1}[2][10] = 10 \]
\[ \text{equ:Equation} \]
\[ \text{equ:Sub} \]
\[ \text{exp:Identifier} \]
\[ \exp:QualifiedNamePart \text{name}="yi1" \]
\[ \exp:ArraySubscripts \]
\[ \exp:IndexExpression \]
\[ \exp:IntegerLiteral 3 \]
\[ \exp:IntegerLiteral 6 \]
\[ \text{equ:Sub} \]
\[ \text{equ:Equation} \]
\[ \text{equ:Sub} \]
\[ \text{exp:Identifier} \]
\[ \exp:QualifiedNamePart \text{name}="xi" \]
\[ \exp:ArraySubscripts \]
\[ \exp:IndexExpression \]
\[ \exp:IntegerLiteral 1 \]
\[ \exp:IntegerLiteral 1 \]
\[ \text{equ:Sub} \]
\[ \text{equ:Equation} \]
\[ \text{equ:Sub} \]
\[ \text{exp:Identifier} \]
\[ \exp:QualifiedNamePart \text{name}="xi" \]
\[ \exp:ArraySubscripts \]
\[ \exp:IndexExpression \]
\[ \exp:IntegerLiteral 2 \]
\[ \exp:IntegerLiteral 5 \]
\[ \text{equ:Sub} \]
\[ \text{equ:Equation} \]
\[ \text{equ:Sub} \]
\[ \text{exp:Identifier} \]
\[ \exp:QualifiedNamePart \text{name}="xi" \]
\[ \exp:ArraySubscripts \]
\[ \exp:IndexExpression \]
\[ \exp:IntegerLiteral 3 \]
\[ \exp:IntegerLiteral 3 \]
\[ \text{equ:Sub} \]
\[ \text{equ:Equation} \]
\[ \text{equ:Sub} \]
\begin{equation}
\begin{align*}
\begin{array}{l}
\text{fun:RecordsList} \\
\text{fun:FunctionList} \\
\text{fun:Function} \\
\hspace{1em} \text{fun:Name} \\
\hspace{2em} \text{fun:QualifiedNamePart name="Cross"} \\
\hspace{2em} \text{fun:QualifiedNamePart name="myCrossInt"} \\
\hspace{1em} \text{fun:OutputVariable type="real"} \\
\hspace{2em} \text{fun:Name} \\
\hspace{3em} \text{fun:QualifiedNamePart name="z"} \\
\hspace{1em} \text{fun:InputVariable type="integer"} \\
\hspace{2em} \text{fun:Name} \\
\hspace{3em} \text{fun:QualifiedNamePart name="x"} \\
\hspace{1em} \text{fun:InputVariable type="integer"} \\
\hspace{2em} \text{fun:Name} \\
\hspace{3em} \text{fun:QualifiedNamePart name="y"} \\
\hspace{1em} \text{fun:InputVariable} \\
\hspace{2em} \text{fun:Assign} \\
\hspace{3em} \text{fun:Identifier} \\
\hspace{4em} \text{exp:QualifiedNamelpart ="ztmp"} \\
\hspace{3em} \text{fun:Identifier} \\
\hspace{2em} \text{fun:Expression} \\
\hspace{3em} \text{exp:Array} \\
\hspace{4em} \text{exp:Sub} \\
\hspace{5em} \text{exp:Mul} \\
\hspace{6em} \text{fun:Identifier} \\
\hspace{7em} \text{exp:QualifiedNamelpart ="x"} \\
\hspace{6em} \text{exp:ArraySubscripts} \\
\hspace{7em} \text{exp:IndexExpression} \\
\hspace{8em} \text{exp:IntegerLiteral} 2 \\
\hspace{7em} \text{exp:IndexExpression} \\
\hspace{6em} \text{exp:ArraySubscripts} \\
\hspace{5em} \text{exp:QualifiedNamelpart} \\
\hspace{4em} \text{fun:Identifier} \\
\hspace{3em} \text{exp:QualifiedNamelpart ="y"} \\
\hspace{3em} \text{exp:ArraySubscripts} \\
\hspace{4em} \text{exp:IndexExpression} \\
\hspace{5em} \text{exp:IntegerLiteral} 3
\end{array}
\end{align*}
\end{equation}
\[
(x_3 y_2 x_3 y_1 x_3)^2
\]
\[
\text{exp:IntegerLiteral}1\text{exp:IntegerLiteral} \\
\text{exp:IndexExpression} \\
\text{exp:ArraySubscripts} \\
\text{exp:QualifiedNamePart} \\
\text{exp:Identifier} \\
\text{exp:QualifiedNamePart name="y">} \\
\text{exp:ArraySubscripts} \\
\text{exp:IndexExpression} \\
\text{exp:IntegerLiteral}3\text{exp:IntegerLiteral} \\
\text{exp:ArraySubscripts} \\
\text{exp:QualifiedNamePart} \\
\text{exp:Identifier} \\
\text{exp:Mul} \\
\text{exp:Sub} \\
\text{exp:Sub} \\
\text{exp:Mul} \\
\text{exp:Identifier} \\
\text{exp:QualifiedNamePart name="x">} \\
\text{exp:ArraySubscripts} \\
\text{exp:IndexExpression} \\
\text{exp:IntegerLiteral}1\text{exp:IntegerLiteral} \\
\text{exp:ArraySubscripts} \\
\text{exp:QualifiedNamePart} \\
\text{exp:Identifier} \\
\text{exp:QualifiedNamePart name="y">} \\
\text{exp:ArraySubscripts} \\
\text{exp:IndexExpression} \\
\text{exp:IntegerLiteral}2\text{exp:IntegerLiteral} \\
\text{exp:ArraySubscripts} \\
\text{exp:QualifiedNamePart} \\
\text{exp:Identifier} \\
\text{exp:QualifiedNamePart name="y">} \\
\text{exp:ArraySubscripts} \\
\text{exp:IndexExpression} \\
\text{exp:IntegerLiteral}1\text{exp:IntegerLiteral}
\]
\[
\begin{align*}
\text{fun:Expression} & \equiv \text{fun:Assign} \\
& \equiv \text{fun:Expression} \\
& \equiv \text{exp:Identifier} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{exp:ArraySubscripts} \\
& \equiv \text{exp:IntegerLiteral} \\
& \equiv \text{exp:ArraySubscripts} \\
& \equiv \text{exp:Identifier} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{exp:IntegerLiteral} \\
& \equiv \text{exp:Identifier} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{exp:IntegerLiteral} \\
& \equiv \text{exp:Identifier} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{fun:Expression} \\
& \equiv \text{fun:Assign} \\
& \equiv \text{fun:Algorithm} \\
& \equiv \text{fun:Function} \\
& \equiv \text{fun:Name} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{exp:QualifiedNamePart} \\
& \equiv \text{fun:Name} \\
\end{align*}
\]
89
\[ \begin{align*}
&\text{exp:Mul} \text{exp:Sub} \\
&\text{exp:Mul} \\
&\text{exp:Mul} \\
&\text{exp:Mul} \\
&\text{exp:Mul} \\
&\text{exp:Mul} \\
&\text{exp:Mul} \text{exp:Sub} \\
&\text{exp:Assignment} \\
&\text{fun:Algorithm} \\
&\text{fun:Function} \\
&\text{fun:FunctionList} \\
&\text{OpenModelicaModelDescription} \\
\end{align*} \]
<xml version="1.0" encoding="UTF-8"?
<OpenModelicaModelDescription>
xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
xmlns:opt="https://svn.jmodelica.org/trunk/XML/daeOptimization.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
fmiVersion="1.0"
modelName="VDP_Opt"
modelIdentifier="VDP_Opt"
guid="{6244967e-9e00-4cb6-9481-39dfb8929efc}"
generationDateAndTime="2012-05-17T01:57:43"
variableNamingConvention="Structured"
numberOfContinuousStates="3"
numberOfEventIndicators="0"
</VendorAnnotations>
<defaultExperiment startTime="0.0" stopTime="1.0" tolerance="1e-006"/>
<ModelVariables>
<ScalarVariable name="x1" valueReference="0" variability="continuous" causality="Internal" alias="noAlias">
<Real start="0.0" fixed="false" />
</QualifiedName>
</ScalarVariable>
<ScalarVariable name="x2" valueReference="1" variability="continuous" causality="Internal" alias="noAlias">
<Real start="1.0" fixed="false" />
</QualifiedName>
</ScalarVariable>
<ScalarVariable name="cost" valueReference="2" variability="continuous" causality="Internal" alias="noAlias">
<Real start="0.0" fixed="false" />
</QualifiedName>
</ScalarVariable>
<ScalarVariable name="der(x1)" valueReference="3" variability="continuous" causality="Internal" alias="noAlias">
</Real>
</QualifiedName>
</ScalarVariable>
</ModelVariables>
<VariableCategory>Derivative</VariableCategory>
<ScalarVariable name="der(x2)
valueReference="4" variability="continuous"
causality="internal" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="x2"/>
  </QualifiedName>
  <VariableCategory>Derivative</VariableCategory>
</ScalarVariable>
<ScalarVariable name="der(cost)
valueReference="5" variability="continuous"
causality="internal" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="cost"/>
  </QualifiedName>
  <VariableCategory>Derivative</VariableCategory>
</ScalarVariable>
<ScalarVariable name="u
valueReference="6" variability="continuous"
causality="input" alias="noAlias">
  <Real />
  <QualifiedName>
    <exp:QualifiedNamePart name="u"/>
  </QualifiedName>
  <VariableCategory>Algebraic</VariableCategory>
</ScalarVariable>
<ScalarVariable name="p1
valueReference="7" variability="parameter"
causality="internal" alias="noAlias">
  <Real start="1.0" fixed="true" />
  <QualifiedName>
    <exp:QualifiedNamePart name="p1"/>
  </QualifiedName>
  <VariableCategory>Parameter</VariableCategory>
</ScalarVariable>
<ScalarVariable name="p2
valueReference="8" variability="parameter"
causality="internal" alias="noAlias">
  <Real start="1.0" fixed="true" />
  <QualifiedName>
    <exp:QualifiedNamePart name="p2"/>
  </QualifiedName>
  <VariableCategory>Parameter</VariableCategory>
</ScalarVariable>
<ScalarVariable name="p3
valueReference="9" variability="parameter"
causality="internal" alias="noAlias">
  <Real start="2.0" fixed="true" />
  <QualifiedName>
    <exp:QualifiedNamePart name="p3"/>
  </QualifiedName>
  <VariableCategory>Parameter</VariableCategory>
</ScalarVariable>
</ModelVariables>
<equ:BindingEquations>
<equ:BindingEquation>
  <equ:Parameter>
    <exp:QualifiedNamePart name="p1"/>
  </equ:Parameter>
  <equ:BindingExp>
    <exp:RealLiteral>1.0</exp:RealLiteral>
  </equ:BindingExp>
</equ:BindingEquation>
<equ:BindingEquation>
  <equ:Parameter>
    <exp:QualifiedNamePart name="p2"/>
  </equ:Parameter>
  <equ:BindingExp>
    <exp:RealLiteral>1.0</exp:RealLiteral>
  </equ:BindingExp>
</equ:BindingEquation>
<equ:BindingEquation>
  <equ:Parameter>
    <exp:QualifiedNamePart name="p3"/>
  </equ:Parameter>
  <equ:BindingExp>
    <exp:RealLiteral>2.0</exp:RealLiteral>
  </equ:BindingExp>
</equ:BindingEquation>
</equ:BindingEquations>
<equ:DynamicEquations>
<equ:Equation>
  <exp:Sub>
    <exp:Der>
      <exp:Identifier>
        <exp:QualifiedNamePart name="cost"/>
      </exp:Identifier>
    </exp:Der>
    <exp:Mul>
      <exp:Identifier>
        <exp:QualifiedNamePart name="p3"/>
      </exp:Identifier>
      <exp:Add>
        <exp:Pow>
          <exp:Identifier>
            <exp:QualifiedNamePart name="x1"/>
          </exp:Identifier>
          <exp:RealLiteral>2.0</exp:RealLiteral>
        </exp:Pow>
        <exp:Add>
          <exp:Pow>
            <exp:Identifier>
              <exp:QualifiedNamePart name="x2"/>
            </exp:Identifier>
            <exp:RealLiteral>2.0</exp:RealLiteral>
          </exp:Pow>
        </exp:Add>
      </exp:Add>
    </exp:Mul>
  </exp:Sub>
</equ:Equation>
</equ:DynamicEquations>
\[ \begin{align*}
\frac{\partial}{\partial x_1} & \left( x_2^2 \cdot p_1 \cdot x_1 \right) \\
& = x_2^2 \cdot p_1 \\
& + \frac{\partial}{\partial x_1} x_2^2 \\
& = x_2^2 \cdot p_1 \\
& + 2x_2 \cdot p_1 \cdot x_1 \\
& + \frac{\partial}{\partial x_1} x_1^2 \\
& = x_2^2 \cdot p_1 \\
& + 2x_2 \cdot p_1 \cdot x_1 \\
& + 2 \cdot 1 \cdot x_1 \\
& = x_2^2 \cdot p_1 \\
& + 2x_2 \cdot p_1 \cdot x_1 \\
& + 2x_1 \\
\end{align*} \]
\begin{equation}
\begin{align*}
x_1 &= 0.0 \\
x_2 &= 1.0 \\
\text{cost} &= 0.0
\end{align*}
\end{equation}

\begin{opt:Optimization}
\begin{opt:ObjectiveFunction}
\begin{exp:TimedVariable}
\begin{exp:Identifier}
\begin{exp:QualifiedNamePart name="cost"/>
\end{exp:Identifier}
\begin{exp:Instant}2.0</exp:Instant>
\end{exp:TimedVariable}
\end{opt:ObjectiveFunction}
\end{opt:Optimization}
B Implementation code

B.1 Template Based XML Code Generator Implementation code

// This file defines templates for transforming flattened Modelica code to XML code.
package CodegenXML
import interface SimCodeTV;

/*************************************
SECTION: SIMULATION TARGET, ROOT TEMPLATE
*************************************/
template translateModel(SimCode simCode)
"Generates root template for compiling a simulation of a Modelica model."
::=
match simCode
case SIMCODE(modelInfo = MODELINFO(__)) then
let() = textFile(generateXml(simCode),
'<?xml version="1.0" encoding="UTF-8"?>
<OpenModelicaModelDescription>
  xmlns:exp="https://svn.jmodelica.org/trunk/XML/daeExpressions.xsd"
  xmlns:equ="https://svn.jmodelica.org/trunk/XML/daeEquations.xsd"
  xmlns:fun="https://svn.jmodelica.org/trunk/XML/daeFunctions.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="daeBase.xsd"
</OpenModelicaModelDescription>

<modelDescriptionXml(simCode, guid)>
<vendorAnnotationsXml(simCode)>
<defaultExperiment(simulationSettingsOpt)>
<modelVariablesXml(modelInfo)>
<bindingEquationsXml(modelInfo)>
<equationsXml(allEquations, whenClauses, helpVarInfo)>
<algorithmicEquationsXml(allEquations)>
<initialEquationsXml(modelInfo)>

<opt:IntervalStartTime>
<opt:Value>0.0</opt:Value>
<opt:Free>false</opt:Free>
<opt:InitialGuess>0.0</opt:InitialGuess>
</opt:IntervalStartTime>
<opt:IntervalFinalTime>
<opt:Value>1.0</opt:Value>
<opt:Free>false</opt:Free>
<opt:InitialGuess>1.0</opt:InitialGuess>
</opt:IntervalFinalTime>
<opt:TimePoints>
<opt:Value>20.0</opt:Value>
</opt:TimePoints>
<opt:Constraints>
<opt:ConstraintLeq>
<exp:Identifier>
<exp:QualifiedNamePart name="u"/>
</exp:Identifier>
<exp:RealLiteral>0.75</exp:RealLiteral>
</opt:ConstraintLeq>
</opt:Constraints>
</opt:Optimization>
</OpenModelicaModelDescription>
}
template vendorAnnotationsXml(SimCode simCode) :
  match simCode
  case SIMCODE(modelInfo = MODELINFO(varInfo = VARINFO(__))) then
    let generationTool = 'OpenModelica Compiler
    <getVersionNr()>'
  <<
  </VendorAnnotations>
</OpenModelicaModelDescription>
end defaultExperiment;

<<
<Tool name="<%generationTool%>"> </Tool>
<SimulationSettings>
</SimulationSettings>
</OpenModelicaModelDescription>
end defaultExperiment;

template modelVariablesXml(ModelInfo modelInfo)
  "Generates XML code for ModelVariables file ."
  ::= match modelInfo
  case MODELINFO(var = SIMVARS(__)) then
    <<
  </ModelVariables>
</OpenModelicaModelDescription>
end modelVariablesXml;

template xsdateTimeXml(DateTime dt)
  "YYYY-MM-DDThh:mm:ss" :
  ::= match dt
case DATETIME(__) then
  "<%year%>-<%twodigit(mon)%>-<%twodigit(day)%>T<%twodigit(hour)%>:<%twodigit(min)%>:<%twodigit(sec)%>">

end xsdateTimeXml;

template
defaultExperiment(Option<SimulationSettings> simulationSettingsOpt)
  "Generates code for defaultExperiment file for FMU target."
  ::= match simulationSettingsOpt
  case SOME(de as SIMULATION_SETTINGS(__)) then
    <<
  </defaultExperiment>
<SimulationSettings>
</SimulationSettings>
</OpenModelicaModelDescription>
end defaultExperiment;

SECTION: GENERATE LINUX FOR MODEL DESCRIPTION

*** SECTION: GENERATE LINUX FOR MODEL DESCRIPTION

* SCALAR VARIABLES IN SIMULATION FILE

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."

"Generates code for defaultExperiment file for FMU target."

"Generates XML code for ModelVariables file ."
case SIMVAR(____) then
<<
<%ScalarVariableAttributesXml(simVar)%>
>>
end ScalarVariableAttributesXml;

template ScalarVariableAttributesXml(SimVar simVar)
"Generates XML code for ScalarVariable Attribute file ."
::= match simVar
case SIMVAR(____) then
let valueReference = '<%System.tmpTick()%>'
let variability = getVariablityXml(varKind)
let description = if comment then
'description="<%Util.escapeModelicaStringToXmlS
tring(comment)%>"'
let alias = getAliasVarXml(aliasvar)
let caus = getCausalityXml(causality)
let variableCategory =
variableCategoryXml(varKind)
<<
<ScalarVariable name="<%crefStrXml(name)%>"
<%description%>
valueReference="<%valueReference%>
variability="<%variability%>
causality="<%caus%>
alias="<%alias%>">
<QualifiedName>
<%qualifiedNamePartXml(name)%>
</QualifiedName>
<VariableCategory><%variableCategory%></VariableCategory>
</ScalarVariable> <%

end ScalarVariableAttributesXml;

template getCausalityXml(Causality c)
"Returns the Causality Attribute of
ScalarVariable."
::= match c
case NONECAUS(____) then "none"
case INTERNAL(____) then "internal"
case OUTPUT(____) then "output"
case INPUT(____) then "input"
end getCausalityXml;

template getVariablityXml(VarKind varKind)
"Returns the variability Attribute of
ScalarVariable."
::= match varKind
case VARIABLE(____) then "Algebraic"
case STATE(____) then "State"
case STATE_DER(____) then "Derivative"
case DUMMY_DER(____) then "Algebraic"
case DUMMY_STATE(____) then "Algebraic"
case DISCRETE(____) then "Algebraic"
case PARAM(____) then "Parameter"
case CONST(____) then "Constant"
else error(sourceInfo(), "Unexpected
simVarTypeName varKind")
end getVariablityXml;

template variableCategoryXml(VarKind varKind)
"Returns the variable category of
ScalarVariable."::= match varKind
case VARIABLE(____) then "Algebraic"
case STATE(____) then "State"
case STATE_DER(____) then "Derivative"
case DUMMY_DER(____) then "Algebraic"
case DUMMY_STATE(____) then "Algebraic"
case DISCRETE(____) then "Algebraic"
case PARAM(____) then "Parameter"
case CONST(____) then "Constant"
else error(sourceInfo(), "Unexpe
ted
simVarTypeName varKind")
end variableCategoryXml;

template ScalarVariableTypeXml(DAE.Type type_,
String unit, String displayUnit,
Option<DAE.Exp> initialValue, Boolean isFixed)
"Generates XML code for ScalarVariable Type
file ."
::= match type_
case T_INTEGER(____) then '<Integer
<%ScalarVariableTypeCommonAttributeXml(initialV
alue,isFixed)%>/>'
case T_REAL(____) then '<Real
<%ScalarVariableTypeCommonAttributeXml(initialV
alue,isFixed)%>
<%ScalarVariableTypeRealAttributeXml(unit,displ
ayUnit)%>/>'
case T_BOOL(____) then '<Boolean
<%ScalarVariableTypeCommonAttributeXml(initialV
alue,isFixed)%>/>'
case T_STRING(____) then '<String
<%ScalarVariableTypeCommonAttributeXml(initialV
alue,isFixed)%>/>'
case T_ENUMERATION(____) then '<Real
<%ScalarVariableTypeCommonAttributeXml(initialV
alue,isFixed)%>/>'
else 'UNKOWN_TYPE'
end ScalarVariableTypeXml;

template ScalarVariableTypeCommonAttributeXml(Option<DAE
.Exp> initialValue, Boolean isFixed)
"Generates XML code for ScalarVariable Type
file ."
::= match initialValue
case SOME(exp) then
'start="<%initValXml(exp)%>">
fixed="<%isFixed%>"
end ScalarVariableTypeCommonAttributeXml;

template initValXml(exp initialvalue)
"Returns initial value of ScalarVariable."
::= match initialvalue
case SOME(exp) then
'start="<%initValXml(exp)%>"
fixed="<%isFixed%>"
end ScalarVariableTypeCommonAttributeXml;

template getAliasVarXml(AliasVariable aliasvar)
"Returns the alias Attribute of
ScalarVariable."
::= match aliasvar
case NOALIAS(____) then "noAlias"
case ALIAS(____) then "alias"
case NEGATEDALIAS(____) then "negatedAlias"
else "noAlias"
end getAliasVarXml;

98
template ScalarVariableTypeRealAttributeXml(String unit, String displayUnit)
"Generates XML code for ScalarVariable Type Real file ."
 ::= 
 let unit_ = if unit then 'unit="<%unit%>"' 
 let displayUnit_ = if displayUnit then 'displayUnit="<%displayUnit%>"'
 <<
 <%unit_%> <%displayUnit_%>
>>
end ScalarVariableTypeRealAttributeXml;

template contextCrefXml(ComponentRef cr, Context context)
"Generates XML code for a component reference depending on which context we're in."
 ::= 
 match context 
 case FUNCTION_CONTEXT(__) then 
 System.unquoteIdentifier(crefStrXml(cr))
 else crefXml(cr)
 end contextCrefXml;

template contextIteratorNameXml(Id name, Context context)
"Generates XML code for an iterator variable."
 ::= 
 match context 
 case FUNCTION_CONTEXT(__) then name
 else name
 end contextIteratorNameXml;

template crefXml(ComponentRef cr)
"Generates XML equivalent name for component reference."
 ::= 
 match cr 
 case CREF_IDENT(iden = "xloc") then 
 crefStrXml(cr)
 case CREF_IDENT(iden = "time") then 
 '<exp:Time>time</exp:Time>'
 case WILD(__) then ''
 else crefToXmlStr(cr)
 end crefXml;

template qualifiedNamePartXml(ComponentRef cr)
"Generates XML code of the Qualified name of a variable . "
 ::= 
 match cr 
 case CREF_IDENT(__) then 
 let arrayTest = arraysubscriptsStrXml(subscriptLst)
 if arrayTest then << 
 <exp:QualifiedNamePart name="<%ident%>">
 %arraysubscriptsStrXml(subscriptList)%
<exp:QualifiedNamePart>
>> else << 
 <exp:QualifiedNamePart name="<%ident%>">
<exp:QualifiedNamePart>
>> else "CREF_NOT_IDENT_OR_QUAL"
 end qualifiedNamePartXml;

template arraysubscriptsStrXml(list<Subscript> subscripts)
"Generates XML code for subscript part of the name."
 ::= 
 if subscripts then 
 <<
<exp:ArraySubscripts>
<%subscripts |> s =>
<exp:IndexExpression>
<exp:IntegerLiteral><%i%></exp:IntegerLiteral>
</exp:IndexExpression>
</exp:ArraySubscripts>
>>
 else 
 "CREF_NOT_IDENT_OR_QUAL"
 end arraysubscriptsStrXml;

template arraysubscriptStrXml(Subscript subscript)
"Generates a single subscript XML code. Only works for constant integer indices."
 ::= 
 match subscript 
 case INDEX(exp=ICONST(integer=i)) then 
 <<
<exp:IndexExpression>
<exp:IntegerLiteral><%i%></exp:IntegerLiteral>
</exp:IndexExpression>
>>
 case SLICE(exp=ICONST(integer=i)) then 
 <<
<exp:IndexExpression>
<exp:IntegerLiteral><%i%></exp:IntegerLiteral>
</exp:IndexExpression>
>>
 case WHOLEDIM(__) then "WHOLEDIM"
 else "UNKNOWN_SUBSCRIPT"
 end arraysubscriptStrXml;

template crefToXmlStr(ComponentRef cr)
"Helper function to crefXml"
 ::= 
 match cr 
 case CREF_IDENT(__) then 
 <<
 <exp:Identifier>
<exp:QualifiedNamePartXml(componentRef)>
</exp:Identifier>
>>
 else 
 <<
 <exp:Identifier>
<exp:QualifiedNamePartXml(componentRef)>
</exp:Identifier>
>>
 end crefToXmlStr;
template crefStrXml(ComponentRef cr)
  "Generates the name of a variable for variable name array."
  ::= match cr
case CREF_IDENT(__) then '
    \<%\qualifiedName\>%\<%subscriptsStrXml\%(\subscriptLst\%)\>%'
case CREF_QUAL(ident = "$DER") then 'der\%(\crefStrXml\%(\componentRef\%)\)%'
case CREF_QUAL(__) then '
    \<%\ident\>%\<%subscriptsStrXml\%(\subscriptLst\%)\%.\<%crefStrXml\%(\componentRef\%)\>%'
  else "CREF_NOT_IDENT_OR_QUAL"
  end crefStrXml;

template crefFunctionNameXml(ComponentRef cr)
  ::= match cr
case CREF_IDENT(__) then System.stringReplace(unquoteIdentifier\%(ident\%)\), "_", "__")
case CREF_QUAL(__) then '
    \%System.stringReplace(unquoteIdentifier\%(ident\%)\), "_", "__")%_<%crefFunctionNameXml\%(\componentRef\%)%>
  end crefFunctionNameXml;

template subscriptStrXml(Subscribe sub)
  "Generates a single subscript."
  Only works for constant integer indexes."
  ::= match sub
  case INDEX\%(exp=ICONST\%(integer=i\%)\)% then i
  case SLICE\%(exp=ICONST\%(integer=i\%)\)% then i
  case WHOLEDIM\%(\%)\% then "WHOLEDIM"
  else "UNKNOWN_SUBSCRIPT"
  end subscriptStrXml;

template expCrefXml(DAE.Exp ecr)
  ::= match ecr
  case CREF\%(\%) then crefXml\%(\componentRef\%)%
case CALL\%(path = IDENT\%(name = "der")\%)
    expLst = \{arg as CREF\%(\%)\}\% then
    <<
      <exp:Der>
        \<%crefXml\%(arg.componentRef\%)%>
      </exp:Der>
    >>
  else "ERROR_NOT_A_CREF"
  end expCrefXml;

template dotPathXml(Path path)
  "Generates paths with components separated by dots."
  ::= match path
  case QUALIFIED\%(\%)\% then '<%name\%>.<%dotPathXml\%(path\%)%>'
case IDENT\%(\%)\% then name
case FULLYQUALIFIED\%(\%)\% then dotPathXml\%(path\%)%
  end dotPathXml;

template underscorePathXml(Path path)
  "Generates XML code for paths."
  ::= match path
  case QUALIFIED\%(\%)\% then '<\%\backslash\%System.stringReplace\%(unquoteIdentifier\%(ident\%)\), ",", ",")%\%<%crefFunctionNameXml\%(\componentRef\%)%>'
  else "ERROR_NOT_A_CREF"
  end underscorePathXml;

template replaceDotAndUnderscoreXml(String str)
  "Replace _ with __ and dot in identifiers with _"  ::= match str
  case name then
    let str_dots = System.stringReplace\%(name\%, ".\", ",\")%/
    let str_underscores = System.stringReplace\%(str_dots\%, ",\", "__")%
    System.unquoteIdentifier\%(str_underscores\%)%
  end replaceDotAndUnderscoreXml;

template substrcptsStrXml(list<Subscript> subscripts)
  "Generates subscript part of the name."
  ::= if subscripts then
    '[\%\$subscripts\% \% s => substrcptsStrXml\%(s\%)\%\% ;separator="\",\%\%]'
case IDENT(__) then
<<
<exp:QualifiedNamePart name="<%name%"/>
>>
case FULLYQUALIFIED(__) then
<<
<%underscorePathXml(path)%>
>>
end underscorePathXml;

/*****************************/
* SECTION: GENERATE All Function IN SIMULATION FILE *
*****************************/
template functionWhenReinitStatementXml(WhenOperator reinit, Text &varDecls /*BUFP*/)
"Generates re-init statement for when equation."
::=
match reinit
  case REINIT(__) then
  let &preExp = buffer "" /*BUFD*/
  let val = daeExpXml(value,
contextSimulationDiscrete,
  &preExp /*BUFC*/, &varDecls /*BUFD*/)
  <<
  <exp:Reinit>
  <%crefXml(stateVar)%>
  <%val%>
  </exp:Reinit>
  >>
  case TERMINATE(__) then
  let &preExp = buffer "" /*BUFD*/
  let msgVar = daeExpXml(message,
contextSimulationDiscrete, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  <<
  <%preExp%>
  <%msgVar%>
  >>
  case ASSERT(source=SOURCE(info=info)) then
  assertCommonXml(condition, message,
contextSimulationDiscrete, &varDecls, info)
  end functionWhenReinitStatementXml;

template functionWhenReinitStatementThenXml(list<WhenOperator> reinits, Text &varDecls /*BUFP*/)
"Generates re-init statement for when equation."
::=
  let body = (reinits |> (reinit =>
    match reinit
      case REINIT(__) then
      let &preExp = buffer "" /*BUFD*/
      let val = daeExpXml(value,
contextSimulationDiscrete,
      &preExp /*BUFC*/, &varDecls /*BUFD*/)
      <<
      <exp:Reinit>
      <%crefXml(stateVar)%>
      <%val%>
      </exp:Reinit>
      >>
      case TERMINATE(__) then
      let &preExp = buffer "" /*BUFD*/
      let msgVar = daeExpXml(message,
contextSimulationDiscrete, &preExp /*BUFC*/, &varDecls /*BUFD*/)
      <<
      <%preExp%>
      <%msgVar%>
      >>
      case ASSERT(source=SOURCE(info=info)) then
      assertCommonXml(condition, message,
contextSimulationDiscrete, &varDecls, info)
      separator="\n"
  %body%)
  >>
end functionWhenReinitStatementThenXml;

template bindingEquationsXml(ModelInfo modelInfo)
"Function for Binding Equations"
::=
match modelInfo
  case MODELINFO(varInfo=VARINFO(numStateVars=numStateVars),vars=SIMVARS(__)) then
  <<
  <equ:BindingEquations>
  <%vars.paramVars |> var =>
  bindingEquationXml(var) ;separator="\n"%
  <%vars.intParamVars |> var =>
  bindingEquationXml(var) ;separator="\n"%
  <%vars.boolParamVars |> var =>
  bindingEquationXml(var) ;separator="\n"%
  <%vars.stringParamVars |> var =>
  bindingEquationXml(var)
  %body%
  >>
  </equ:BindingEquations>
end bindingEquationsXml;

template bindingEquationXml(SimVar var)
"Generate XML code for binding Equations"
::=
match var
  case SIMVAR(__) then
  let varName = '<%qualifiedNamePartXml(name)%>'
  match initialValue
  case SOME(exp) then
"
let &varDecls = buffer "" /*BUFD*/
let &preExp = buffer "" /*BUFD*/
<<
<equ:BindingEquation>
<equ:Parameter>
<%varName%>
</equ:Parameter>
<equ:BindingExp>
</equ:BindingEquation>

end bindingEquationXml;

template equationsXml( list<SimEqSystem> allEquationsPlusWhen, list<SimWhenClause> whenClauses, list<HelpVarInfo> helpVarInfo)
"Function for all equations"
::=
let &varDecls = buffer "" /*BUFD*/
let jens = System.tmpTickReset(0)
let &tmp = buffer ""
let eqs = (allEquationsPlusWhen |> eq =>
equation_Xml(eq, contextSimulationDiscrete, &varDecls /*BUFD*/, &tmp);
separator="\n")
let reinit = (whenClauses |>
when hasindex i0 =>
generateReinitXml(when, &varDecls,i0)>
genreinitXmlsXml(when, &varDecls,i0)
_FIXVERIFIER<%eqs%>
_FIXVERIFIER<%reinit%>
end equationsXml;

template algorithmicEquationsXml( list<SimEqSystem> allEquations)
"Generates XML for an equation that is an algorithm."
::=
let &varDecs = buffer "" /*BUFD*/
let algs = (allEquations |> eq =>
equationAlgorithmXml(eq, contextSimulationDiscrete, &varDecs /*BUFD*/, &tmp);
separator="\n")
<<
<fun:Algorithm>
<%algs%>
</fun:Algorithm>
end algorithmicEquationsXml;

template equationAlgorithmXml(SimEqSystem eq, Context context, Text &varDecs /*BUFP*/)
"Generates XML for an equation that is an algorithm."
::=
match eq
case SES_ALGORITHM(__) then
let alg = (statements |> stmt =>
algorithmStatementXml(stmt, contextFunction, &varDecs /*BUFD*/)
;separator="\n")
<<
<fun:Algorithm>
<%alg%>
</fun:Algorithm>
end equationAlgorithmXml;

template initialEquationsXml(ModelInfo modelInfo)
"Function for Initial Equations."
::=
match modelInfo
case MODELINFO(varInfo=VARINFO(numStateVars=numStateVars), vars=SIMVARS(__)) then
<%vars.stateVars |> var =>
initialEquationXml(var) ;separator="\n"%>
<%vars.derivativeVars |> var =>
initialEquationXml(var) ;separator="\n"%>
<%vars.intAlgVars |> var =>
initialEquationXml(var) ;separator="\n"%>
<%vars.boolAlgVars |> var =>
initialEquationXml(var) ;separator="\n"%>
<%vars.stringAlgVars |> var =>
initialEquationXml(var) ;separator="\n"%>
end initialEquationsXml;

template initialEquationXml(SimVar var)
"Generates XML code for Initial Equations."
::=
match var
case SIMVAR(__) then
let identName = '<%crefXml(name)%>'
match initialValue
case SOME(exp) then
let &varDecls = buffer "" /*BUFD*/
let &preExp = buffer "" /*BUFD*/
<<
<equ:Equation>
<equ:Sub>
<%identName%>
<%daeExpXml(exp, contextOther, &preExp, &varDecls)%>
</equ:Sub>
</equ:Equation>
_FIXVERIFIER<%n%>
</equ:DyamicEquations>
_FIXVERIFIER<%tmp%>
_FIXVERIFIER<%eqs%>
_FIXVERIFIER<%reinit%>
end initialEquationsXml;

/*************************************
* SECTION: GENERATE All EQUATIONS IN SIMULATION FILE
**************
-------------------------------------------------------------

/********************
* SECTION: GENERATE All EQUATIONS IN SIMULATION FILE
*
* Generates an equation.
* This template should not be used for a SES_RESIDUAL.
* Residual equations are handled differently.
* ------------------------------------------
* match context case INLINE_CONTEXT() then
* old_equation_Xml(eq,context,Text &varDecs /*BUFP*/,Text &eqs)
* "Generates an equation.
* This template should not be used for a SES_RESIDUAL.
* Residual equations are handled differently."
* match context case INLINE_CONTEXT() then
* old_equation_Xml(eq,context,Text &varDecs /*BUFP*/,Text &eqs)
* "Generates an equation.
* This template should not be used for a SES_RESIDUAL.
* Residual equations are handled differently."
function System.tmpTickIndex(10)
    let ix = System.tmpTickIndex(10)
    let &tmp = buffer ""
    let &varD = buffer ""
    let x = match eq
        case e as SES_SIMPLE_ASSIGN(__)
            then equationSimpleAssignXml(e, context, &varD /*BUFD*/)
        case e as SES_ARRAY_CALL_ASSIGN(__)
            then equationArrayCallAssignXml(e, context, &varD /*BUFD*/)
        case e as SES_LINEAR(__)
            then equationLinearXml(e, context, &varD /*BUFD*/)
        case e as SES_NONLINEAR(__)
            then equationNonlinearXml(e, context, &varD /*BUFD*/)
        case e as SES_WHEN(__)
            then "NOT IMPLEMENTED EQUATION"
            else "NOT IMPLEMENTED EQUATION"
            let &eqs += <<
                %x
            >> end equation_Xml;

template old_equation_Xml(SimEqSystem eq, Context context, Text &varDecls)
    "Generates an equation.
    This template should not be used for a SES_RESIDUAL.
    Residual equations are handled differently."
    ::= match eq
        case e as SES_MIXED(__)
            then equationSimpleAssignXml(e, context, &varDecls)
        case e as SES_SIMPLE_ASSIGN(__)
            then equationArrayCallAssignXml(e, context, &varDecls)
        case e as SES_ARRAY_CALL_ASSIGN(__)
            then equationLinearXml(e, context, &varDecls)
        case e as SES_LINEAR(__)
            then equationNonlinearXml(e, context, &varDecls)
        case e as SES_NONLINEAR(__)
            then equationWhenXml(e, context, &varDecls)
        case e as SES_WHEN(__)
            then "NOT IMPLEMENTED EQUATION"
            else "NOT IMPLEMENTED EQUATION"
            end old_equation_Xml;

template equationSimpleAssignXml(SimEqSystem eq, Context context, Text &varDecls)
    "Generates an equation that is just a simple assignment."
    ::= match eq
        case SES_SIMPLE_ASSIGN(__)
            then let &preExp = buffer ""
            let expPart = daeExpXml(eq.exp, context, &preExp /*BUFD*/)
            let &helpInits = buffer ""
            let helpIf = (conditions |> (e, hidx) =>
                        let helpInit = daeExpXml(e, context, &preExp /*BUFC*/), &varDecls /*BUFD*/)
                        |
                        let &helpInits += ' <%helpInit%' ; separator=" || ")
                match expTypeFromExpShortXml(eq.exp)
                case "boolean" then <<
                case "integer" then <<
                case "real" then <<
                else error(sourceInfo(), 'No runtime support for this sort of array call: %printExpStr(eq.exp)')
                >>
                end equationSimpleAssignXml;

template equationArrayCallAssignXml(SimEqSystem eq, Context context, Text &varDecls)
    "Generates equation on form 'cref_array = call(...)'."
    ::= <<
        match eqn as SES_ARRAY_CALL_ASSIGN(__) then
            let &preExp = buffer ""
            let expPart = daeExpXml(eqn.exp, context, &preExp /*BUFD*/)
            let &helpInits = buffer ""
            let helpIf = (conditions |> (e, hidx) =>
                        let helpInit = daeExpXml(e, context, &preExp /*BUFC*/), &varDecls /*BUFD*/)
                        |
                        let &helpInits += ' <%helpInit%' ; separator=" || ")
                match expTypeFromExpShortXml(eqn.exp)
                case "boolean" then <<
                case "integer" then <<
                case "real" then <<
                else error(sourceInfo(), 'No runtime support for this sort of array call: %printExpStr(eqn.exp)')
                >>
                end equationArrayCallAssignXml;

template equationLinearXml(SimEqSystem eq, Context context, Text &varDecls)
    "Generates a linear equation system."
    ::= match eq
        case SES_LINEAR(__)
            then let uid = System.tmpTick()
            let size = listLength(vars)
            let aname = 'A<%uid%>'
            let bname = 'b<%uid%>'
            let mixedPostfix = if partOfMixed then "_mixed" //else ""
            <<
            simJac |> (row, col, eq as SES_RESIDUAL(__)) =>
                let &preExp = buffer ""
                let expPart = daeExpXml(eq.exp, context, &preExp /*BUFC*/), &varDecls /*BUFD*/)
                \"%preExp%\"%expPart\" ; separator=" \n"
            beqs |> exp hasindex i0 =>
                let &preExp = buffer ""
                let expPart = daeExpXml(eq.exp, context, &preExp /*BUFC*/), &varDecls /*BUFD*/)
                \"%preExp%\"%expPart\" ; separator=" \n"
            &eqs += '<%crefXml(cref)%> %expPart%>
        >>
        end equationLinearXml;

template equationWhenXml(SimEqSystem eq, Context context, Text &varDecls)
    "Generates an equation that is just a simple assignment."
    ::= match eq
        case SES_WHEN(__)
            then equationWhenXml(e, context, &varDecls)
        else "NOT IMPLEMENTED EQUATION"

error(sourceInfo(), 'No runtime support for this sort of array call: %printExpStr(eqn.exp)')

end equation_Xml;
template equationLinearXml(SimEqSystem eq, Context context, Text &varDecls /*BUFP*/, Text &tmp)
"Generates a mixed equation system."
::= match eq case SES_MIXED(____) then let conteqs = equation_Xml(cont, context, &varDecls /*BUFD*/, &tmp)
let numDiscVarsStr = listLength(discVars)
let valueLenStr = listLength(values)
let &preDisc = buffer "" /*BUFD*/
let discLoc2 = (discEqs |> (discEqs |>
SSES_SIMPLE_ASSIGN(__) hasindex i0 =>
let expPart = daeExpXml(exp, context, &preDisc /*BUFC*/, &varDecls /*BUFD*/)
<< <\crefXml(name)> = <\expPart>; discrete_loc2[<%i0%>] = <\crefXml(name)>;
>> ;separator="n"
)
<< #ifdef _OMC_MEASURE_TIME
SIM_PROF_TICK_EQ(SIM_PROF_EQ_<%index%>);
#endif
mixed_equation_system(<%numDiscVarsStr%>);
modelica_boolean values[<%numDiscVarsStr%>];
int value_dims[<%numDiscVarsStr%>];
<%discVars |> SIMVAR(__) hasindex i0 =>
'discrete_loc[<%i0%>] = <\crefXml(name)>';
;separator="n"
{
  { conteqs%
  }
<preDisc%>
 \<Discloc2%>
  {
    modelica_boolean *loc_ptrs[<%numDiscVarsStr%>] = {<%discVars |>
SIMVAR(__) =>
'(modelica_boolean*)&<\crefXml(name)>'
;separator=""};
  check_discrete_values(<%numDiscVarsStr%>, valuesLenStr%);
}
mixed_equation_system_end(<%numDiscVarsStr%>);
#ifdef _OMC_MEASURE_TIME
SIM_PROF_ACC_EQ(SIM_PROF_EQ_<%index%>);
#endif
end equationLinearXml;

template equationMixedXml(SimEqSystem eq, Context context, Text &varDecls /*BUFP*/, Text &tmp)
"Generates a mixed equation system."
::= match eq case SES_MIXED(____) then let conteqs = equation_Xml(cont, context, &varDecls /*BUFD*/, &tmp)
let numDiscVarsStr = listLength(discVars)
let valueLenStr = listLength(values)
let &preDisc = buffer "" /*BUFD*/
let discLoc2 = (discEqs |> (discEqs |>
SSES_SIMPLE_ASSIGN(__) hasindex i0 =>
let expPart = daeExpXml(exp, context, &preDisc /*BUFC*/, &varDecls /*BUFD*/)
<< <\crefXml(name)> = <\expPart>; discrete_loc2[<%i0%>] = <\crefXml(name)>;
>> ;separator="n"
)
<< #ifdef _OMC_MEASURE_TIME
SIM_PROF_TICK_EQ(SIM_PROF_EQ_<%index%>);
#endif
mixed_equation_system(<%numDiscVarsStr%>);
modelica_boolean values[<%numDiscVarsStr%>];
int value_dims[<%numDiscVarsStr%>];
<%discVars |> SIMVAR(__) hasindex i0 =>
'discrete_loc[<%i0%>] = <\crefXml(name)>';
;separator="n"
{
  { conteqs%
  }
<preDisc%>
 \<Discloc2%>
  {
    modelica_boolean *loc_ptrs[<%numDiscVarsStr%>] = {<%discVars |>
SIMVAR(__) =>
'(modelica_boolean*)&<\crefXml(name)>'
;separator=""};
  check_discrete_values(<%numDiscVarsStr%>, valuesLenStr%);
}
mixed_equation_system_end(<%numDiscVarsStr%>);
#ifdef _OMC_MEASURE_TIME
SIM_PROF_ACC_EQ(SIM_PROF_EQ_<%index%>);
#endif
end equationMixedXml;

template equationNonlinearXml(SimEqSystem eq, Context context, Text &varDecls /*BUFP*/)
"Generates a non linear equation system."
::= match eq case SES_NONLINEAR(____) then let size = listLength(crefs)
<< crefs |> name hasindex i0 =>
<\crefXml(name)>;
<<<%crefXml(name)>%>

<< #ifdef _OMC_MEASURE_TIME
SIM_PROF_TICK_EQ(SIM_PROF_EQ_<%index%>);
#endif
end equationNonlinearXml;

template equationWhenXml(SimEqSystem eq, Context context, Text &varDecls /*BUFP*/)
"Generates a when equation XML."
::= match eq case SES_WHEN(left=left, right=right, conditions=conditions, elseWhen = NONE()) then let &preExp = buffer "" /*BUFD*/
let &helpInits = buffer "" /*BUFD*/
let helpIf = (conditions |> (e, hidx) =>
 let helpInit = daeExpXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
<< <\crefXml(name)> = <\exp>;\
delete _loc[<%hidx%>] = <\crefXml(name)>;
>> ;separator="n"
)
let &preExp2 = buffer "" /*BUFD*/
let exp = daeExpXml(right, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
let elseWhen = equationElseWhenXml(elseWhenEq, context, preExp, helpInits, varDecls)
let cond = if preExp then preExp else helpInits
<<
<equ:When>
<equ:Condition>
<%cond%>
<equ:Equation>
<exp:Sub>
<%crefXml(left)%>
<%exp>
<\exp:Sub>
</equ:Equation>
</equ:When>
>>
case SES_WHEN(left=left, right=right, conditions=conditions, elseWhen = SOME(elseWhenEq)) then let &preExp = buffer "" /*BUFD*/
let &helpInits = buffer "" /*BUFD*/
let helpIf = (conditions |> (e, hidx) =>
 let helpInit = daeExpXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
<< <\crefXml(name)> = <\exp>;\
delete _loc[<%hidx%>] = <\crefXml(name)>;
>> ;separator="n"
)
let &preExp2 = buffer "" /*BUFD*/
let exp = daeExpXml(right, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
let elseWhen = equationElseWhenXml(elseWhenEq, context, preExp, helpInits, varDecls)
let cond = if preExp then preExp else helpInits
<<
<equ:When>
<equ:Condition>
<%cond%>
<equ:Equation>
<exp:Sub>
<%crefXml(left)%>
<%exp>
<\exp:Sub>
</equ:Equation>
</equ:When>
>>
end equationWhenXml;
template equationElseWhenXml(SimEqSystem eq, Context context, Text &preExp /*BUFD*/, Text &helpInits /*BUFD*/, Text &varDecls /*BUFD*/)
"Generates a else when equation."
::= match eq
   case SES_WHEN(left=left,
     right=right,conditions=conditions,elseWhen = NONE()) then
     let helpIf = conditions |> 
       (e, hidx) => let helpInit = daeExpXml(e, context, 
       &preExp /*BUFC*/, &varDecls /*BUFD*/)
       let &helpInits += '<%helpInit%>'
       ;separator=" || ")
     let &preExp2 = buffer " /*BUFD*/
     let exp = daeExpXml(right, context, &preExp2 /*BUFC*/,
     &varDecls /*BUFD*/)
     let cond = if preExp then preExp else
       helpInits
     <<
     <equ:ElseWhen>
     <equ:Condition>
     <%cond%>
     </equ:Condition>
     <equ:Equation>
     <exp:Sub>
     <%crefXml(left)%>
     <%exp%>
     </exp:Sub>
     </equ:Equation>
     </equ:ElseWhen>
     >>
   end equationElseWhenXml;

/*****************************/
* SECTION: GENERATE ALL RECORDS ( RECORD LIST) IN SIMULATION FILE
/*****************************/
template recordDeclarationXml(RecordDeclaration recDecl)
"Generates XML structs for a record declaration."
::= match recDecl
   case RECORD_DECL_FULL(__) then
     <<
     <fun:Record>
     <fun:Name>
     <exp:QualifiedNamePart
     name='<%name%>'/>
     </fun:Name>
     <%variables |> var  => recordBodyXml(var)
     ;separator=" || ">
     </fun:Record>
     >>
   case RECORD_DECL_DEF(__) then
     Record Declaration definition is left empty for testing
     >>
   end recordDeclarationXml;
template recordBodyXml(Variable var)
::= match var
   case VARIABLE(__) then
     <<
     <fun:Field type="Record">
     <fun:Name>
     <exp:QualifiedNamePart
     name='<%contextCrefXml(name,contextFunction)%>'/>
     </fun:Name>
     <fun:Record>
     <%varTypeXml(var)%>
     </fun:Record>
     </fun:Field>
     >>
   end recordBodyXml;

/*****************************/
* SECTION: GENERATE All USER DEFINED FUNCT INCLUDING EXTERNAL FUNCTIONS IN SIMULATION FILE
/*****************************/
template functionsXml(list<Function> functions)
"Generates the body for a set of functions."
::= <<
   <fun:FunctionList>
   <%functions |> fn => functionXml(fn)
   ;separator="\n">%>
   </fun:FunctionList>
   >>
end functionsXml;

template functionXml(Function fn)
"Generates the body for a function."
::=
match fn
case fn as FUNCTION(_)
  then regularFunctionXml(fn)
case fn as EXTERNAL_FUNCTION(_)
  then externalFunctionXml(fn)
case fn as RECORD_CONSTRUCTOR(_)
  then ''
end functionXml;

template regularFunctionXml(Function fn)
"Generates XML code for a Modelica function."
::=
match fn
case FUNCTION(_)
  let()= System.tmpTickReset(1)
  let fname = underscorePathXml(name)
  let &varDecls = buffer "" /"BUFD="/ 
  let bodyPart = (body |> stmt  
    =>

funStatementXml(stmt, &varDecls /*BUFD*/)
);

  separator="\n"
<<
</fun:Function>
</fun:Name>
</%fname%>
<%outVars |> var |
  letOutputVariableXml(var) ;separator="\n"%>
<%funArgs |> var |
  letArgDefinitionXml(var) ;separator="\n"%>
<fun:Algorithm>
<%bodyPart%>
</fun:Algorithm>
</fun:Function>

end regularFunctionXml;

template externalFunctionXml(Function fn)
"Generates the body for an external function (just a wrapper)."
::=
match fn
case FUNCTION(_)
  let()= System[tmpTickReset(1)]
  let &preExp = buffer "" /"BUFD="/ 
  let &varDecls = buffer "" /"BUFD="/ 
  let name = underscorePathXml(name)
  let callPart = extFunCallXml(fn, &preExp /*BUFD*/, &varDecls /*BUFD*/)
<<
</fun:Function>
</fun:Name>
</%fname%>
<%outVars |>
  letOutputVariableXml(var) ;separator="\n"%>
<%funArgs |>
  letArgDefinitionXml(var) ;separator="\n"%>
<fun:Algorithm>
<%callPart%>
</fun:Algorithm>
</fun:Function>

end externalFunctionXml;

template funArgNameXml(Variable var)
::=
match var
case VARIABLE(_)
  contextCrefXml(name,contextFunction)
case FUNCTION_PTR(_)
  name
end funArgNameXml;

template letOutputVariableXml(Variable var)
::=
match var
case VARIABLE(ty=T_COMPLEX(complexClassType=RECORD(_))
  <<
<fun:OutputVariable type="Record">
<fun:Name>
<exp:QualifiedNamePart name="%contextCrefXml(name,contextFunction)%"/>
</fun:Name>
<fun:Record>
<%varTypeXml(var)%>
<fun:Record>
</fun:OutputVariable>

end letOutputVariableXml;

template letArgDefinitionXml(Variable var)
::=
match var
case VARIABLE(ty=T_COMPLEX(complexClassType=RECORD(_))
  <<
<fun:InputVariable type="Record">
<fun:Name>
<exp:QualifiedNamePart name="%contextCrefXml(name,contextFunction)%"/>
<fun:Record>
<%varTypeXml(var)%>
<fun:Record>
</fun:InputVariable>

end letArgDefinitionXml;

// test code ...

template funVarDeclarationsXml(Variable var) ::= 
    match var 
    case VARIABLE(__) then 
        <fun:protectedVariable 
          type="<%varTypeXml(var)%>">
            <fun:Name>
                <exp:QualifiedNamePart 
                    name="<%contextCrefXml(name,contextFunction)%>"/>
            </fun:Name>
            <%/*underscorePathXml(ClassInf.getStateName(complexClassType))*/%>
        </fun:ProtectedVariable>
    end funVarDeclarationsXml;

template extFunctionNameXml(String name, String language) ::= 
    match language 
    case "C" then 
        <exp:QualifiedName name="<%name%>/">
    case "FORTRAN 77" then 
        <exp:QualifiedName name="<%name%>/">
    else error(sourceInfo(), 'Unsupport external language: <%language%>')
    end extFunctionNameXml;

template extTypeXml(Type type, Boolean isInput, Boolean isArray) 
    "Generates type for external function argument or return value."
    ::= 
        let s = match type 
        case T_INTEGER(__) then "int" 
        case T_REAL(__) then "double" 
        case T_STRING(__) then "const char*" 
        case T_BOOL(__) then "int" 
        case T_ENUMERATION(__) then "int" 
        case T_ARRAY(__) then extTypeF77Xml(ty, isInput, true) 
        case T_COMPLEX(complexClassType=EXTERNAL_OBJ(__)) 
            then "void*" 
        case T_COMPLEX(complexClassType=RECORD(path=rname)) 
            then 'struct <%underscorePathXml(rname)%>' 
        case T_METATYPE(__) case T_METABOXED(__) then "void*" 
        else error(sourceInfo(), 'Unknown external C type <%typeString(type)%>') 
        match type case T_ARRAY(__) then s else if isInput then (if isArray then '<%match s case "const char*" then "*" else "const %><%s%>*' else s) else '<%s%>*' 
        end extTypeXml;

template extTypeF77Xml(Type type, Boolean isReference) 
    "Generates type for external function argument or return value for F77."
    ::= 
        let s = match type 
        case T_INTEGER(__) then "int" 
        case T_REAL(__) then "double" 
        case T_STRING(__) then "char" 
        case T_BOOL(__) then "int" 
        case T_ENUMERATION(__) then "int" 
        case T_ARRAY(__) then extTypeF77Xml(ty, true) 
        case T_COMPLEX(complexClassType=EXTERNAL_OBJ(__)) 
            then "void*" 
        end extTypeF77Xml;

template functionNameXml(Function fn, Boolean dotPath) ::= 
    match fn 
    case FUNCTION(__) then 
    case EXTERNAL_FUNCTION(__) then 
    case RECORD_CONSTRUCTOR(__) then if dotPath 
        then dotPathXml(name) else 
        underscorePathXml(name) 
    end functionNameXml;

template extVarNameXml(ComponentRef cr) ::= 
    <<<%crefXml(cr)%>>
    end extVarNameXml;

template extFunCallXml(Function fun, Text &preExp /*BUFP*/, Text &varDecls /*BUFD*/) 
    "Generates the call to an external function." 
    ::= 
        match fun 
        case EXTERNAL_FUNCTION(__) then 
        case RECORD_CONSTRUCTOR(__) then if dotPath 
            then dotPathXml(name) else 
            underscorePathXml(name) 
        end extFunCallXml;

template extFunCallCXml(Function fun, Text &preExp /*BUFC*/, Text &varDecls /*BUFD*/) 
    "Generates the call to an external C function." 
    ::= 
        match fun 
        case EXTERNAL_FUNCTION(__) then 
        match language 
            case "C" then extFunCallCXml(fun, &preExp /*BUFC*/, &varDecls /*BUFD*/)
        "Generates the call to an external function." 
            ::= 
                match fun 
                case EXTERNAL_FUNCTION(__) then 
                match language 
                    case "C" then extFunCallCXml(fun, &preExp /*BUFC*/, &varDecls /*BUFD*/)
                    case "FORTRAN 77" then extFunCallF77Xml(fun, &preExp /*BUFC*/, &varDecls /*BUFD*/)
                end extFunCallCXml;

template extFunCallF77Xml(Function fun, Text &preExp /*BUFP*/, Text &varDecls /*BUFD*/) 
    "Generates the call to an external C function." 
    ::= 
        match fun 
        case EXTERNAL_FUNCTION(__) then 
            let args = (extArgs |> arg => extArgCXml(arg, &preExp /*BUFC*/, &varDecls /*BUFD*/) ;separator="\n") 
            let returnAssign = match extReturn case 
                SIMEXTARG(cref=c) then 
                    '<%extVarNameXml(c)%> ' 
                else 
                    ""
            <<
                <fun:Assign>
                    <%returnAssign%>
                    <exp:Expression>
                        <exp:FunctionCall>
                            <exp:Name>
    end extFunCallF77Xml;
</exp:QualifiedNamePart>
  name="<%extName%>">
  </exp:Name>
  <exp:Arguments>
  <%args%>
  </exp:Arguments>
  </exp:FunctionCall>
</exp:Expression>
</fun:Assign>
>
end extFunCallCXml;


template extFunCallF77Xml(Function fun, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates the call to an external Fortran 77 function."
::=
  match fun
case EXTERNAL_FUNCTION(__) then
    let args = (extArgs |> arg =>
    extArgF77Xml(arg, &preExp, &varDecls);
    separator=", ")
    let returnAssign = match extReturn case
    SIME<brarg(cref=c) then
      '<%extVarNameXml(c)%>'
    else
      ""
    <<
    <fun:Assign>
    <%returnAssign%>
    <exp:Expression>
    <exp:FunctionCall>
      <exp:Name>
      <exp:QualifiedNamePart
      name="<%extName%>">
      </exp:Name>
      <exp:Arguments>
      <%args%>
      </exp:Arguments>
      </exp:FunctionCall>
    </exp:Expression>
    </fun:Assign>
  >>
end extFunCallF77Xml;

template extArgCXml(SimExtArg extArg, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Helper to extFunCallXml."
::=
  match extArg
case SIMEXTARG(cref=c, isArray=true, type_=t) then
    <<
    <%extVarNameXml(c)%>
  >>
case SIMEXTARG(cref=c, isInput=ii, outputIndex=0, type_=T_INTEGER(__)) then
    <<
    <%extVarNameXml(c)%>
  >>
case SIMEXTARG(cref=c, isInput=ii, outputIndex=oi, type_=T_STRING(__)) then
    <<
    <%extVarNameXml(c)%>
  >>
case SIMEXTARG(cref=c, isArray=false, type_=t) then
    <<
    <%extVarNameXml(c)%>
  >>
case SIMEXTARGEXP(__) then
    let texp = daeExpXml(exp, contextFunction, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    <<
    <%texp%>
  >>
case SIMEXTARGSIZE(cref=c) then
    let dim = daeExpXml(exp, contextFunction, &preExp, &varDecls)
    let name = extVarNameXml(c)
    <<
    <exp:Size>
    <%name%>
    <%dim%>
    </exp:size>
  >>
end extArgCXml;

/**
* SECTION: GENERATE OPTIMIZATION IN SIMULATION FILE
***************************************************************************/

template optimizationXml( list<DAE.Constraint> constraints)
"Generates XML for Optimization."
::=
  (constraints |> constraint =>
    constraintsXml(constraint); separator="\n")
end optimizationXml;

template constraintsXml(Constraint cons)
"Generates XML for List of Constraints."
::=
  (cons |> constraint =>
    constraintXml(constraint); separator="\n")
end constraintsXml;

template extArgF77Xml(SimExtArg extArg, Text &preExp, Text &varDecls)
::=
  match extArg
case SIMEXTARG(cref=c, isArray=true, type_=t) then
    <<
    <NextVarNameXml(c)%>
  >>
case SIMEXTARG(cref=c, outputIndex=oi, type_=T_INTEGER(__)) then
    <<
    <NextVarNameXml(c)%>
  >>
case SIMEXTARGEXP(exp=exp, type_=T_STRING(__)) then
    let tepx = daeExpXml(exp, contextFunction, &preExp /*BUFC*/, &varDecls /*BUFD*/) +'
<!-- external xml -->
    <<
    <%tep%>"test
  >>
case SIMEXTARGSIZE(cref=c) then
    let dim = daeExpXml(exp, contextFunction, &preExp, &varDecls)
    let name = extVarNameXml(c)
    <<
    <exp:size>
    <%name%>
    <%dim%>
    </exp:size>
  >>
end extArgF77Xml;
;separator="\n")
<<
<opt:Optimization>
<opt:Constraints>
</opt:Constraints>
</opt:Optimization>
>>
else error(sourceInfo(), 'Unknown Constraint List')
end constraintsXml;

*************************************
*         SECTION: GENERATE All Algorithm IN SIMULATION FILE
*************************************
template funStatementXml(Statement stmt, Text &varDecls /*BUFP*/)
  "Generates function statements."
  ::= match stmt case ALGORITHM(__) then algStatementXml(stmt, contextFunction, &varDecls /*BUFD*/)
  ;separator="\n")
else "NOT IMPLEMENTED FUN STATEMENT"
end funStatementXml;

template algStatementXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
  "Generates an algorithm statement."
  ::= let res = match stmt case STMT_ASSIGN(__) then algStmtAssignXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_ASSIGN_ARR(__) then algStmtAssignArrXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_TUPLE_ASSIGN(__) then algStmtTupleAssignXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_IF(__) then algStmtIfXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_FOR(__) then algStmtForXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_WHILE(__) then algStmtWhileXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_ASSERT(__) then algStmtAssertXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_TERMINATE(__) then algStmtTerminateXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_WHEN(__) then algStmtWhenXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_BREAK(__) then algStmtBreakXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_RETURN(__) then algStmtReturnXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_NORETCALL(__) then algStmtNoretcallXml(stmt, context, &varDecls /*BUFD*/)
  case STMT_REINIT(__) then algStmtReinitXml(stmt, context, &varDecls /*BUFD*/)
  else error(sourceInfo(), 'ALG_STATEMENT NYI')
end algStatementXml;

template algStmtAssignXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
  "Generates an assignment algorithm statement."
  ::= match stmt case STMT_ASSIGN(exp1=CREF(componentRef=WILD(__)), exp=e) then let &preExp = buffer "" /*BUFD*/
  let expPart = daeExpXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  <fun:Assign>
    <%expPart%>
  </fun:Expression>
  </fun:Assign>
end algStmtAssignXml;

template algStmtAssignArrXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
  "Generates an assignment algorithm statement for arrays."
  ::= match stmt case STMT_ASSIGN(exp1=CREF(ty = T_FUNCTION_REFERENCE_VAR(__))) then let &preExp = buffer "" /*BUFD*/
  let varPart = scalarLhsCrefXml(exp1, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  let expPart = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  <fun:Assign>
    <%varPart%>
    <fun:Expression>
      <%expPart%>
    </fun:Expression>
  </fun:Assign>
end algStmtAssignArrXml;

template algStmtTupleAssignXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
  "Generates an assignment algorithm statement for tuples."
  ::= match stmt case STMT_ASSIGN(exp1=CREF(__)) then let &preExp = buffer "" /*BUFD*/
  let varPart = scalarLhsCrefXml(exp1, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  let expPart = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  <fun:Assign>
    <%varPart%>
    <fun:Expression>
      <%expPart%>
    </fun:Expression>
  </fun:Assign>
end algStmtTupleAssignXml;

109
let expPart = daeExpXml(val, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
<<
<fun:Assign>
  <%varPart%>
  <fun:Expression>
    <%expPart%>
  </fun:Expression>
</fun:Assign>
>>
}
case STMT_ASSIGN(__) then
let &preExp = buffer "" /*BUFD*/
let expPart1 = daeExpXml(exp1, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)
let expPart2 = daeExpXml(exp, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)
<<
<fun:Assign>
  <%expPart1%>
  <fun:Expression>
    <%expPart2%>
  </fun:Expression>
</fun:Assign>
>>
end algStmtAssignXml;

template algStmtAssignArrXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates an array assignment algorithm statement."
::=
match stmt
case STMT_ASSIGN_ARR(exp=e, componentRef=cr, type_=t) then
let &preExp = buffer "" /*BUFD*/
let expPart = daeExpXml(e, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)
let ispec = indexSpecFromCrefXml(cr, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)
if ispec then
<<
<%preExp%>
<%indexedAssignXml(t, expPart, cr, ispec, context, &varDecls)%>
>>
else
<<
<fun:Assign>
  <%expPart%>
  <fun:Expression>
    <%expPart%>
  </fun:Expression>
</fun:Assign>
>>
end algStmtAssignArrXml;

template indexedAssignXml(DAE.Type ty, String exp, DAE.ComponentRef cr, String ispec, Context context, Text &varDecls)
::=
let type = expTypeArrayXml(ty)
let cref = contextArrayCrefXml(cr, context)
match context
case FUNCTION_CONTEXT(__) then '
<<
<%cref%>
>>
end indexedAssignXml;

template copyArrayDataXml(DAE.Type ty, String exp, DAE.ComponentRef cr, Context context)
::=
let type = expTypeArrayXml(ty)
let cref = contextArrayCrefXml(cr, context)
match context
case FUNCTION_CONTEXT(__) then '
<<
<%cref%>
>>
end copyArrayDataXml;

template algStmtTupleAssignXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates XML for a tuple assignment algorithm statement."
::=
match stmt
case STMT_TUPLE_ASSIGN(exp=CALL(__)) then
let &preExp = buffer "" /*BUFD*/
let &afterExp = buffer "" /*BUFD*/
let crefs = (expExpLst |> e =>
  ExpressionDump.printExpStr(e) ;separator="",
  if marker then
    '('
  else
    ''
)'
let marker = '('
<<
<%cref%>
</fun:FunctionCallStatement>
else
<<
<%exp%>
<%ispec%>
<%cref%>
</fun:FunctionCallStatement>
>>
end algStmtTupleAssignXml;

template indexedAssignXml(DAE.Type ty, String exp, DAE.ComponentRef cr, String ispec, Context context, Text &varDecls)
::=
let type = expTypeArrayXml(ty)
let cref = contextArrayCrefXml(cr, context)
match context
case FUNCTIONCONTEXT(__) then '
<<
<%cref%>
>>
end indexedAssignXml;

template copyArrayDataXml(DAE.Type ty, String exp, DAE.ComponentRef cr, Context context)
::=
let type = expTypeArrayXml(ty)
let cref = contextArrayCrefXml(cr, context)
match context
case FUNCTIONCONTEXT(__) then '
<<
<%cref%>
>>
end copyArrayDataXml;

let iterName = contextIteratorNameXml(iterator, context)
let &preExp = buffer ""
let startValue = daeExpXml(start, context, &preExp, &varDecs)
let stepValue = match step case SOME(eo) then
daеExpXml(eo, context, &preExp, &varDecs)
else
  "" //because the default step value is 1
let stopValue = daeExpXml(stop, context, &preExp, &varDecs)
end algStmtForRange_implXml;

template algStmtForGenericXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates a for algorithm statement where range is not RANGE."
::= match stmt
case STMT_FOR(__) then
  let iterType = expTypeXml(type_, iterIsArray)
  let arrayType = expTypeArrayXml(type_)
  let stmtStr = (statementList | > stmt =>
    algStatementXml(stmt, context, &varDecls ;separator="\n")
  )
end algStmtForGeneric_xml;

template algStmtForGeneric_implXml(Exp exp, Ident iterator, String type, String arrayType, Boolean iterIsArray, Text &body, Context context, Text &varDecs)
"The implementation of algStmtForGeneric, which is also used by daeExpReduction."
::= let iterName = contextIteratorNameXml(iterator, context)
  let &preExp = buffer ""
  let evar = daeExpXml(exp, context, &preExp, &varDecs)
end algStmtForGeneric_implXml;

template algStmtWhileXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates a while algorithm statement."
::= match stmt
case STMT_WHILE(__) then
  let &preExp = buffer "" /*BUFD*/
  let var = daeExpXml(exp, context, &preExp /*BUFC*/ , &varDecs /*BUFD*/)
  <<
    <fun:While>
      <fun:Condition>
        <%var%>
      </fun:Condition>
      <fun:Statements>
        <%statementList |> stmt =>
          algStatementXml(stmt, context, &varDecs /*BUFD*/) ;separator="\n"%>
      </fun:Statements>
    </fun:While>
  >>
end algStmtWhileXml;

template algStmtAssertXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates an assert algorithm statement."
::= match stmt
case STMT_ASSERT(source=SOURCE(info=info)) then
  assertCommonXml(cond, msg, context, &varDecls, info)
end algStmtAssertXml;

template algStmtTerminateXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates a terminate algorithm statement."
::= match stmt
case STMT_TERMINATE(__) then
  let &preExp = buffer "" /*BUFD*/
  let msgVar = daeExpXml(msg, context, &preExp /*BUFC*/ , &varDecs /*BUFD*/)
  <<
    <%preExp%>
    <%msgVar%>
  >>
end algStmtTerminateXml;

template algStmtNoretcallXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
"Generates a no return call algorithm statement."
::= match stmt
case STMT_NORETCALL(__) then
  let &preExp = buffer "" /*BUFD*/
  let expPart = daeExpXml(exp, context, &preExp /*BUFC*/ , &varDecs /*BUFD*/)
  <<
    <%preExp%>
    <%expPart%>
  >>
end algStmtNoretcallXml;
template algStmtWhenXml(DAE.Statement when, Context context, Text &varDecls /*BUFP*/)
  "Generates a when algorithm statement."
  ::= match context
case SIMULATION(genDiscrete=true) then
  match when
  case STMT_WHEN(_) then
    let preIf = algStatementWhenPreXml(when, &varDecls /*BUFD*/)
    let statements = (when.statementLst |> stmt =>
      algStatementXml(stmt, context, &varDecls /*BUFD*/);
    separator="\n")
    let else = algStatementWhenElseXml(when.elseWhen, &varDecls /*BUFD*/)
    <<
      <fun:When>
      <fun:Condition>
        <%preIf%>
      <fun:Condition>
      <fun:Statements>
      <%statements%>
      <fun:Statements>
      <%else%>
    >>
  end match
end algStmtWhenXml;

.template algStatementWhenPreXml(DAE.Statement stmt, Text &varDecls /*BUFP*/)
  "Helper to algStmtWhen."
  ::= match stmt
case STMT_WHEN(exp=ARRAY(array=el)) then
    let restPre = match elseWhen case SOME(ew) then
      algStatementWhenPreXml(ew, &varDecls /*BUFD*/)
    else
      ""
    let &preExp = buffer "" /*BUFD*/
    let assignments =
      algStatementWhenPreAssignsXml(el, helpVarIndices, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    <<
      <%preExp%>
      <%restPre%>
    >>
  case when as STMT_WHEN(_)
    match helpVarIndices
    case {[i] then
      let restPre = match when.elseWhen case SOME(ew)
      algStatementWhenPreXml(ew, &varDecls /*BUFD*/)
    }""
    let &preExp = buffer "" /*BUFD*/
    let res = daeExpXml(when.exp, contextSimulationDiscrete, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    <<
      <%preExp%>
      <%restPre%>
    >>
  end algStatementWhenPreAssignsXml;

template algStmtReinitXml(DAE.Statement stmt, Context context, Text &varDecls /*BUFP*/)
  "Generates an assignement algorithm statement."
  ::= match stmt
case STMT_REINIT(_) then
  <<
    <fun:When>
    <fun:Condition>
      <%firstInt%>
    <fun:Condition>
    <fun:Statements>
    <%firstExpPart%> <!--This exp matches the first exp -->
    <fun:Statements>
    <%else%>
    <fun:When>
  >>
end algStmtReinitXml;


let &preExp = buffer "" /*BUFD*/
let expPart1 = daeExpXml(var, context, 
&preExp /*BUFC*/, &varDecls /*BUFD*/)
let expPart2 = daeExpXml(value, context, 
&preExp /*BUFC*/, &varDecls /*BUFD*/)
<<
<exp:Reinit>
  <expPart1%>
  <expPart2%>
<<
</exp:Reinit>
end algStmtReinitXml;

template indexSpecFromCrefXml(ComponentRef cr, 
Context context, Text &preExp /*BUFP*/, 
Text &varDecls /*BUFP*/)
  "Helper to algStmtAssignArr. Currently works only for CREF_IDENT."
::= 
match cr 
case CREF_IDENT(subscriptLst=subs as (_ :: _)) then 
  daeExpCrefRhsIndexSpecXml(subs, context, 
  &preExp /*BUFC*/, &varDecls /*BUFD*/)
end indexSpecFromCrefXml;

template elseExprXml(DAE.Else else_, Context 
context, Text &varDecls /*BUFP*/)
  "Helper to algStmtIf."
::= 
match else_ 
case NOELSE(__) then 
  ""
  case ELSEIF(__) then 
    let &preExp = buffer "" /*BUFD*/
    let condExp = daeExpXml(exp, context, 
    &preExp /*BUFC*/, &varDecls /*BUFD*/)
    <<
      <fun:ElseIf>
        <fun:Condition>
          <%condExp%>
        </fun:Condition>
        <%statementLst |> stmt 
         =>algStatementXml(stmt, context, &varDecls /*BUFD*/);separator="\n"%>
      </fun:ElseIf>
    >>
    case ELSEIF(___ then 
      let &preExp = buffer "" /*BUFD*/
      let condExp = daeExpXml(exp, context, 
      &preExp /*BUFC*/, &varDecls /*BUFD*/)
    <<
      <fun:ElseIf>
        <fun:Condition>
          <%condExp%>
        </fun:Condition>
        <%statementLst |> stmt 
         =>algStatementXml(stmt, context, &varDecls /*BUFD*/);separator="\n"%>
      </fun:ElseIf>
    >>
    else 
      daeExpBinaryXml(exp, context, &preExp /*BUFC*/, 
      &varDecls /*BUFD*/);
    esac
  esac
end elseExprXml;

template scalarLhsCrefXml(Exp ecr, Context 
context, Text &preExp, Text &varDecls)
  "Generates the left hand side (for use on left 
hand side) of a component 
reference."
::= 
match ecr 
case CREF(componentRef = cr, ty = T_FUNCTION_REFERENCE_VAR(__)) then 
  if crefNoSub(ecr.componentRef) then 
    crefXml(ecr.componentRef)
  else 
    daeExpCrefLhsXml(ecr, context, &preExp, 
    &varDecls)
  case ecr as CREF(componentRef=CREF_QUAL(__)) then 
    crefXml(ecr.componentRef)
  case ecr as CREF(componentRef=WILD(__)) then 
    "ONLY_IDENT_OR_QUAL_CREF_SUPPORTED_LHSS"
end scalarLhsCrefXml;

/************************************* 
*         SECTION: GENERATE All DAE 
Expression IN SIMULATION FILE 
*************************************/
template daeExpXml(Exp exp, Context context, 
Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
  "Root Template for Expression-XML generation."
::= 
match exp 
case e as ICONST(__) then 
  '<exp:IntegerLiteral><%integer%></exp:IntegerLiteral>'
  case e as RCONST(__) then 
  '<exp:RealLiteral><%real%></exp:RealLiteral>'
  case e as SCONST(__) then 
  '<exp:StringLiteral><%daeExpSconstXml(string, 
   context, &preExp /*BUFC*/, &varDecls /*BUFD*/)%></exp:StringLiteral>'
  case e as BCONST(__) then 
  '<exp:BooleanLiteral><%if bool then "1" else 
   "0"%> <exp:BooleanLiteral>'
  case e as ENUM_LITERAL(__) then index 
  case e as CREF(__) then 
    daeExpCrefRhsXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as BINARY(__) then 
    daeExpBinaryXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as UNARY(__) then 
    daeExpUnaryXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as LBINARY(__) then 
    daeExpLbinaryXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as LUNARY(__) then 
    daeExpLunaryXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as RELATION(__) then 
    daeExpRelationXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as IFEXP(__) then 
    daeExpIfXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as CALL(__) then 
    daeExpCallXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as ARRAY(__) then 
    daeExpArrayXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as MATRIX(__) then 
    daeExpMatrixXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as RANGE(__) then 
    daeExpRangeXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
  case e as CAST(__) then 
    daeExpCastXml(e, context, &preExp /*BUFC*/, 
    &varDecls /*BUFD*/)
end daeExpXml;
case e as ASUB(____)            then daeExpAsubXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  case e as TSUB(____)            then 'daeExpTsubXml(e, context, &preExp, &varDecls)%-targ1'
  case e as SIZE(____)            then daeExpSizeXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  case e as BOX(____)             then daeExpBoxXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  case e as UNBOX(____)           then daeExpUnboxXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
  case e as SHARED_LITERAL(____)  then daeExpSharedLitXml(e, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
else error(sourceInfo(), 'Unknown expression: <%ExpressionDump.printExpStr(exp)%>')</end daeExpXml;
template daeExternalXmlExp(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
  "Like daeExp," ::=
  match typeof(exp) case T_ARRAY(____) then  // Array
    <<
      <%daeExpXml(exp, context, &preExp, &varDecls)%>
    >>
else daeExpExternalXml;
template daeExpSconstXml(String string, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
  "Generates code for a string constant."
  ::=
  let box = daeExpCrefRhsArrayBoxXml(ecr, context, &preExp, &varDecls)
  if box then box
  else if crefIsScalar(ecr) then
    "<%crefXml(ecr)>
  else if crefSubIsScalar(ecr) then
    // The array subscript results in a scalar
    "<%Util.escapeModelicaStringToCString(ecr)%>
  else daeExpCrefRhs2Xml(ecr, context, &preExp, &varDecls)
end daeExpCrefRhsXml;
template daeExpCrefRhs2Xml(Exp ecr, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
  "Generates code for a component reference on the right hand side of an expression."
  ::=
  match ecr case ecr as CREF(componentRef=cr, ty=ty) then
    let &preExp += '/* daeExpCrefRhs2 begin
      preExp (<%ExpressionDump.printExpStr(ecr)%>) */
    let box = daeExpCrefRhsArrayBoxXml(ecr, context, &preExp, &varDecls)
    if box then
      box
    else if crefIsScalar(ecr) then
      "<%crefXml(ecr.componentRef)%>
    else if crefSubIsScalar(ecr) then
      // The array subscript results in a scalar
      "<%Util.escapeModelicaStringToCString(ecr)%>
    else daeExpCrefRhs2Xml(ecr, context, &preExp, &varDecls)
end daeExpCrefRhsXml;
\[
\text{match crefLastType(cr) case et as T\_ARRAY(__) then <<}
\]
\[
(&<\text{arrName}>)[<\text{threadDimSubListXml(et.dims,cr,context,&preExp,&varDecls)>]}
\]
\[
\text{else error(sourceInfo(),'Indexing non-array}<\text{printExpStr(ecr)}>')
\]
\[
\text{else // The array subscript denotes a slice}
\]
\[
\text{// let &preExp += '/* daeExpCrefRhs2 SLICE(<\text{ExpressionDump.printExpStr(ecr)>})}
\]
\[
\text{let arrName = contextArrayCrefXml(cr,context)}
\]
\[
\text{let arrType = expTypeInfoArrayXml(ty)}
\]
\[
\text{let tmp = tempDeclXml(arrType, &varDecls /*BUFD*/)}
\]
\[
\text{let spec1 = daeExpCrefRhsIndexSpecXml(crefSubs(cr),context, &preExp /*BUFP*/, &varDecls /*BUFD*/)}
\]
\[
\text{let &preExp += <<}
\]
\[
<\text{arrName}>>%>)%>\text{match sub}
\]
\[
\text{case INDEX(__) then let expPart = daeExpXmlElement(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)}
\]
\[
\text{let str = <<}
\]
\[
<\text{expPart}>}
\]
\[
\text{str}
\]
\[
\text{case WHOLEDIM(__) then let str = <<(1), (int*)0, 'W'>> str case SLICE(__) then}
\]
\[
\text{let expPart = daeExpXmlElement(exp, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)}
\]
\[
\text{let tmp = tempDeclXml("modelica_integer", &varDecls /*BUFD*/)}
\]
\[
\text{let &preExp += 'create_index_spec(&<tmp>, nridx_str, idx_str);<}
\]
\[
\text{tmp}
\]
\[
\text{end daeExpCrefRhsIndexSpecXml;}
\]
\[
\text{template daeExpCrefRhsArrayBoxXml(Exp ecr, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)}
\]
\[
\text{"Helper to daeExpCrefRhs." := let nridx_str = listLength(subs)}
\]
\[
\text{let idx_str = (subs |> sub =>}
\]
\[
\text{match sub}
\]
\[
\text{case {} then "" else '<%threadDimSubListXml(dimrest,subrest,context,&preExp,&varDecls)>%>'}}
\]
\[
\text{else error(sourceInfo(),'Less subscripts that dimensions in indexing cref? That's odd!')}
\]
\[
\text{else error(sourceInfo(),'Non-index subscript in indexing cref? That's odd!')}
\]
\[
\text{end threadDimSubListXml;}
\]
\[
\text{template daeExpCrefRhsIndexSpecXml(list<Subscript> subs, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)}
\]
\[
\text{"Helper to daeExpCrefRhs." := let nridx_str = listLength(subs)}
\]
\[
\text{let idx_str = (subs |> sub =>}
\]
\[
\text{match sub}
\]
\[
\text{case INDEX(__) then let expPart = daeExpXmlElement(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)}
\]
\[
\text{let str = <<}
\]
\[
<\text{expPart}>}
\]
\[
\text{str}
\]
\[
\text{case WHOLEDIM(__) then let str = <<(1), (int*)0, 'W'>> str case SLICE(__) then}
\]
\[
\text{let expPart = daeExpXmlElement(exp, context, &preExp /*BUFC*/ , &varDecls /*BUFD*/)}
\]
\[
\text{let tmp = tempDeclXml("modelica_integer", &varDecls /*BUFD*/)}
\]
\[
\text{let &preExp += 'create_index_spec(&<tmp>, nridx_str, idx_str);<}
\]
\[
\text{tmp}
\]
\[
\text{end daeExpCrefRhsIndexSpecXml;}
\]
\[
\text{template daeExpCrefRhsArrayBoxXml(Exp ecr, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)}
\]
\[
\text{"Helper to daeExpCrefRhs." := let nridx_str = listLength(subs)}
\]
\[
\text{let idx_str = (subs |> sub =>}
\]
\[
\text{match sub}
\]
\[
\text{case {} then "" else '<%threadDimSubListXml(dimrest,subrest,context,&preExp,&varDecls)>%>'}}
\]
\[
\text{else error(sourceInfo(),'Less subscripts that dimensions in indexing cref? That's odd!')}
\]
\[
\text{else error(sourceInfo(),'Non-index subscript in indexing cref? That's odd!')}
\]
\[
\text{end threadDimSubListXml;}
\]
let &preExp +=
  <<
<<%arrayCrefXmlStr(ecr.componentRef)>%>>
end daeExpCrefRhsArrayBoxXml;

template daeExpRecordCrefRhsXml(DAE.Type ty, ComponentRef cr, Context context, Text &preExp /*BUFP*/,
  Text &varDecls /*BUFD*/)
::= match ty
case T_COMPLEX(complexClassType = record_state, varList = var_lst) then
  let vars = var_lst |> v =>
    daeExpXml(makeCrefRecordExp(cr,v), context, &preExp, &varDecls)
  ;separator="\n"
<<%vars%>>
end daeExpRecordCrefRhsXml;

**********************************************************************************
* LEFT HAND SIDE
**********************************************************************************
template daeExpCrefLhsXml(Exp exp, Context context, Text &afterExp /*BUFP*/, Text &varDecls /*BUFD*/)
"Generates code for a component reference on the left hand side of an expression."
::= match exp
  case exp as CREF(componentRef=cr, ty=ty) then
    daeExpCrefLhs2Xml(exp, context, &afterExp /*BUFC*/, &varDecls /*BUFD*/)
  else daeExpCrefLhs2Xml(exp, context, &afterExp /*BUFC*/, &varDecls /*BUFD*/)
end daeExpCrefLhsXml;

template daeExpCrefLhs2Xml(Exp ecr, Context context, Text &afterExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates XML code for a component reference on the left hand side!"
::= match ecr
  case ecr as CREF(componentRef=cr, ty=ty) then
    let &afterExp += '/* daeExpCrefLhs2begin
    afterExp <<%ExpressionDump.printExpStr(ecr)%;*/
    let box = daeExpCrefLhsArrayBoxXml(ecr, context, &afterExp, &varDecis)
    if box then
      box
    else if crefIsScalar(cr, context) then
      <<%contextCrefXml(cr,context)%>>
    else if crefSubIsScalar(cr) then
      // The array subscript results in a scalar
      let &afterExp += '/* daeExpCrefLhs2
      SCALAR(<%ExpressionDump.printExpStr(ecr)%>)
      afterExp */<%\n%>'
      let arrName = contextCrefXml(crefStripLastSubs(cr), context)
      let arrayType = expTypeArrayXml(ty)
      let dimLenStr = listLength(crefSubs(cr))
      let dimValuesStr = (crefSubs(cr) |> INDEX(__) =>
        daeExpXml(exp, context, &afterExp /*BUFC*/, &varDecIs /*BUFD*/)
      ;separator="\n")
      match arrayType
      case "metatype_array" then
        'arrayGet(<%arrName%>,<%dimValuesStr%>) /*DAE.CREF */'
      else
        <<<exp:Identifier>
          <exp:QualifiedNamePart
            name="<%arrName%>">
            <exp:ArraySubscripts>
              <exp:IndexExpression>
                <%dimValuesStr%>
            </exp:IndexExpression>
        </exp:ArraySubscripts>
        </exp:QualifiedNamePart>
      </exp:Identifier>
      else
        // The array subscript denotes a slice
        let &afterExp += '/* daeExpCrefLhs2
        SLICE(<%ExpressionDump.printExpStr(ecr)%>)
        afterExp */<%\n%>'
        let arrName = contextArrayCrefXml(cr, context)
        let arrayType = expTypeArrayXml(ty)
        let tmp = tempDeclXml(arrayType, &varDecIs /*BUFD*/)
        let spec1 =
          daeExpCrefLhsIndexSpecXml(crefSubs(cr), context, &afterExp /*BUFC*/, &varDecIs /*BUFD*/)
        let &afterExp +=
          'indexed_assign_<%arrayType%>(&<%tmp%>, &<%arrName%>, &<%spec1%>);<%\n%>'
        tmp
    case ecr then
    let &afterExp += '/* daeExpCrefLhs2 UNHANDLED(<%ExpressionDump.printExpStr(ecr)%>)
    afterExp */<%\n%>'
end daeExpCrefLhs2Xml;
/* SimCodeC.tpl template: daeExpCrefLhs2:
UNHANDLED EXPRESSION:
* <%ExpressionDump.printExpStr(ecr)%>
*/

template daeExpCrefLhs2Xml;

daeExpCrefLhsIndexSpecXml(list<Subscript> subs, Context context, Text &afterExp /*BUFP*/, Text &varDecls /*BUFP*/) /*BUFP*/;
"Helper to daeExpCrefLhs.
::=
let nridx_str = listLength(subs)
let idx_str = (subs |> sub =>
match sub
  case INDEX(__) then
    let expPart = daeExpXml(exp, context, &afterExp /*BUFC*/, &varDecls /*BUFD*/), make_index_array(1, (int) <%expPart%>, 'S') str
  case WHOLEDIM(__) then
    let str = <<(1), (int*)0, 'W'>>
  case SLICE(__) then
    let expPart = daeExpXml(exp, context, &afterExp /*BUFC*/, &varDecls /*BUFD*/)
    let tmp = tempDeclXml("modelica_integer", &varDecls /*BUFD*/)
    let &afterExp += '<%tmp%> = size_of_dimension_integer_array(<%expPart%>, 1);<%n%>
    let str = integer_array_make_index_array(&<%expPart%>, 'A'>
    separator="",
  )
let &afterExp += 'create_index_spec(&<%tmp%>, <%nridx_str%>, <%idx_str%>);<%n%>

end daeExpCrefLhsIndexSpecXml;

daeExpCrefLhsArrayBoxXml(Exp ecr, Context context, Text &afterExp /*BUFP*/, Text &varDecls /*BUFP*/) /*BUFP*/;
"Helper to daeExpCrefLhs.
::=
match ecr as
  CREF(ty=T_ARRAY(ty=aty,dims=dims))
  then
    let tmpArr = tempDeclXml("modelica_integer", &varDecls /*BUFD*/), make_index_array(1, (int)ATY(aty), 'S') str
    let tmp = tempDeclXml(expTypeArrayXml(aty, dims), &varDecls /*BUFD*/)
    let &afterExp += '<%tmp%> = %tmpArr%);<%n%>
    let dimsLenStr = listLength(dims)
    let dimsValuesStr = (dims |> dim =>
      dimensionXml(dim) ;separator="",
    )
    let type = expTypeShortXml(aty)
    &afterExp += '<%type%>_array_create(&%tmp%>)

end daeExpCrefLhsArrayBoxXml;

daeExpRecordCrefLhsXml(DAE.Type ty, ComponentRef cr, Context context, Text &afterExp /*BUFP*/ /*BUFP*/), Text &varDecls /*BUFP*/;
::=
madeInCrefRecordExp(cr, v) then
  let record_type_name = underscorePathXml(ClassInf.getStateName(record_state))
  let ret_type = '<%record_type_name%>_rettype'
  let ret_var = tempDeclXml(ret_type, &varDecls /*BUFD*/)
  let &afterExp += '<%ret_var%> = %ret_var%);<%n%>

end daeExpRecordCrefLhsXml;

daeExpBinaryXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/ /*BUFP*/;
"Generates code for a binary expression.
::=
match exp
  case BINARY(__) then
    let e1 = daeExpXml(exp1, context, &preExp /*BUFC*/, &varDecls /*BUFD*/), e2 = daeExpXml(exp2, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    let operator
      case ADD(__) then
      case SUB(__) then
      case MUL(__) then
      case DIV(__) then
      case POW(__) then

end daeExpBinaryXml;
case UMINUS(__) then daeExpUnaryXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
case ADD_ARR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case SUB_ARR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case MUL_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case ADD_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for ADD_ARRAY_SCALAR"
case SUB_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for SUB_SCALAR_ARRAY"
case MUL_SCALAR_PRODUCT(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_scalar"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_scalar"
case MUL_MATRIX_PRODUCT(__) then
  let typeShort = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer"
case DIV_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case DIV_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for DIV_SCALAR_ARRAY"
case DIV_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for DIV_ARRAY_SCALAR"
case POW_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for POW_ARRAY_SCALAR"
case POW_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for POW SCALAR_ARRAY"
case MUL_ARRAY(__) then 'daeExpBinary:ERR for MUL ARR'
case DIV_ARR(__) then 'daeExpBinary:ERR for DIV ARR'
case MUL_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case MUL_MATRIX_PRODUCT(__) then
  let typeShort = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer"
case DIV_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case DIV_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for DIV_SCALAR_ARRAY"
case DIV_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for DIV_ARRAY_SCALAR"
case POW_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for POW_ARRAY_SCALAR"
case POW_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for POW SCALAR_ARRAY"
case MUL_ARRAY(__) then 'daeExpBinary:ERR for MUL ARR'
case DIV_ARR(__) then 'daeExpBinary:ERR for DIV ARR'
case MUL_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case MUL_MATRIX_PRODUCT(__) then
  let typeShort = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer"
case DIV_ARRAY_SCALAR(__) then
  let type = match ty case
  T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
case T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
case DIV_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for DIV_SCALAR_ARRAY"
case DIV_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for DIV_ARRAY_SCALAR"
case POW_ARRAY_SCALAR(__) then
  "daeExpBinary:ERR for POW_ARRAY_SCALAR"
case POW_SCALAR_ARRAY(__) then
  "daeExpBinary:ERR for POW SCALAR_ARRAY"
case UMINUS_ARR(__) then
error(sourceInfo(),"unary minus for non-real
arrays not implemented")
else error(sourceInfo(),"daeUnary:ERR")
end daeUnaryXML;

template daeExpBinaryXml(Exp exp, Context
context, Text &preExp /*BUFP*/,
    Text &varDecls /*BUFD*/)
"Generates code for a logical binary
expression."
::=
match exp
case LBINARY(__) then
    let e1 = daeExpXml(exp1, context, &preExp
    /*BUFC*/, &varDecls /*BUFD*/)
    let e2 = daeExpXml(exp2, context, &preExp
    /*BUFC*/, &varDecls /*BUFD*/)
    match operator
    case AND(__) then
        <<
<exp:And>
    <%e1%>
    <%e2%>
</exp:And>
>>
case OR(__) then
    <<
<exp:Or>
    <%e1%>
    <%e2%>
</exp:Or>
>>
else "daeExpBinary:ERR"
end daeExpBinaryXml;

template daeExpUnaryXml(Exp exp, Context
context, Text &preExp /*BUFP*/,
    Text &varDecls /*BUFD*/)
"Generates code for a logical unary
expression."
::=
match exp
case LUNARY(__) then
    let e = daeExpXml(exp, context, &preExp
    /*BUFC*/, &varDecls /*BUFD*/)
    match operator
    case NOT(__) then
        <<
<exp:Not>
    <%e%>
</exp:Not>
>>
case GREATER(|ty = T_BOOL(__)|) then
    '(!<%e1%> & & <%e2%>)'
case GREATER(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case GREATER(__) then
    <<
<exp:LogGt>
    <%e1%>
    <%e2%>
</exp:LogGt>
>>
case GREATEREQ(|ty = T_BOOL(__)|) then
    '(!<%e1%> || !<%e2%>)'
case GREATEREQ(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case GREATEREQ(__) then
    <<
<exp:LogGeq>
    <%e1%>
    <%e2%>
</exp:LogGeq>
>>
case LESS(|ty = T_BOOL(__)|) then
    '(!<%e1%> & & <%e2%>)'
case LESS(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case LESS(__) then
    <<
<exp:LogLt>
    <%e1%>
    <%e2%>
</exp:LogLt>
>>
case LESSEQ(|ty = T_BOOL(__)|) then
    '(!<%e1%> || <%e2%>)'
case LESSEQ(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case LESSEQ(__) then
    <<
<exp:LogLeq>
    <%e1%>
    <%e2%>
</exp:LogLeq>
>>
case GREATEREQ(|ty = T_BOOL(__)|) then
    '(!<%e1%> || !<%e2%>)'
case GREATEREQ(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case GREATEREQ(__) then
    <<
<exp:LogGeq>
    <%e1%>
    <%e2%>
</exp:LogGeq>
>>
case EQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && !<%e2%>) || (<%e1%> && <%e2%>))'
case EQUAL(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case EQUAL(__) then
    <<
<exp:LogEq>
    <%e1%>
    <%e2%>
</exp:LogEq>
>>
case NEQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && <%e2%>) || (<%e1%> && !<%e2%>))'
case NEQUAL(|ty = T_STRING(__)|) then
    '!stringEqual(<%e1%>, <%e2%>))'
case NEQUAL(__) then
    <<
<exp:LogNeq>
    <%e1%>
    <%e2%>
</exp:LogNeq>
>>
else "daeExpRelation:ERR"
end daeExpRelationXml;

template daeExpRelationSimXml(Exp exp, Context
context, Text &preExp /*BUFP*/,
    Text &varDecls /*BUFD*/)
"Generates code for a relation expression."
::=
match exp
case rel as RELATION(__) then
    let simRel = daeExpRelationSimXml(rel,
    context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    if simRel then
        simRel
    else
        let e1 = daeExpXml(rel.exp1, context, &preExp
        /*BUFC*/, &varDecls /*BUFD*/)
        let e2 = daeExpXml(rel.exp2, context, &preExp
        /*BUFC*/, &varDecls /*BUFD*/)
        match rel.operator
        case LESS(|ty = T_BOOL(__)|) then
            '(!<%e1%> & & <%e2%>)'
case LESS(|ty = T_STRING(__)|) then
            '(!stringCompare(<%e1%>, <%e2%>) < 0)'
case LESS(__) then
            <<
<exp:LogLt>
    <%e1%>
    <%e2%>
</exp:LogLt>
>>
case GREATER(|ty = T_BOOL(__)|) then
    '(!<%e1%> & & !<%e2%>)'
case GREATER(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) > 0)'
case GREATER(__) then
    <<
<exp:LogGt>
    <%e1%>
    <%e2%>
</exp:LogGt>
>>
case LESSEQ(|ty = T_BOOL(__)|) then
    '(!<%e1%> || <%e2%>)'
case LESSEQ(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) <= 0)'
case LESSEQ(__) then
    <<
<exp:LogLeq>
    <%e1%>
    <%e2%>
</exp:LogLeq>
>>
case GREATEREQ(|ty = T_BOOL(__)|) then
    '(<%e1%> || !<%e2%>)'
case GREATEREQ(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) >= 0)'
case GREATEREQ(__) then
    <<
<exp:LogGeq>
    <%e1%>
    <%e2%>
</exp:LogGeq>
>>
case EQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && !<%e2%>) || (<%e1%> && <%e2%>))'
case EQUAL(|ty = T_STRING(__)|) then
    '!stringEqual(<%e1%>, <%e2%>))'
case EQUAL(__) then
    <<
<exp:LogEq>
    <%e1%>
    <%e2%>
</exp:LogEq>
>>
case NEQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && %<e2%>) || (>%e1%> && !%e2%))'
case NEQUAL(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case NEQUAL(__) then
    <<
<exp:LogNeq>
    <%e1%>
    <%e2%>
</exp:LogNeq>
>>
else "daeExpRelation:ERR"
end daeExpRelationXml;

template daeExpRelationXml(Exp exp, Context
context, Text &preExp /*BUFP*/,
    Text &varDecls /*BUFD*/)
"Generates code for a relation expression."
::=
match exp
case rel as RELATION(__) then
    let simRel = daeExpRelationSimXml(rel,
    context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    if simRel then
        simRel
    else
        let e1 = daeExpXml(rel.exp1, context, &preExp
        /*BUFC*/, &varDecls /*BUFD*/)
        let e2 = daeExpXml(rel.exp2, context, &preExp
        /*BUFC*/, &varDecls /*BUFD*/)
        match rel.operator
        case LESS(|ty = T_BOOL(__)|) then
            '(!<%e1%> & & <%e2%>)'
case LESS(|ty = T_STRING(__)|) then
            '(!stringCompare(<%e1%>, <%e2%>) < 0)'
case LESS(__) then
            <<
<exp:LogLt>
    <%e1%>
    <%e2%>
</exp:LogLt>
>>
case GREATER(|ty = T_BOOL(__)|) then
    '(!<%e1%> & & !<%e2%>)'
case GREATER(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) > 0)'
case GREATER(__) then
    <<
<exp:LogGt>
    <%e1%>
    <%e2%>
</exp:LogGt>
>>
case LESSEQ(|ty = T_BOOL(__)|) then
    '(!<%e1%> || <%e2%>)'
case LESSEQ(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) <= 0)'
case LESSEQ(__) then
    <<
<exp:LogLeq>
    <%e1%>
    <%e2%>
</exp:LogLeq>
>>
case GREATEREQ(|ty = T_BOOL(__)|) then
    '(<%e1%> || !<%e2%>)'
case GREATEREQ(|ty = T_STRING(__)|) then
    '(!stringCompare(<%e1%>, <%e2%>) >= 0)'
case GREATEREQ(__) then
    <<
<exp:LogGeq>
    <%e1%>
    <%e2%>
</exp:LogGeq>
>>
case EQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && !<%e2%>) || (<%e1%> && <%e2%>))'
case EQUAL(|ty = T_STRING(__)|) then
    '!stringEqual(<%e1%>, <%e2%>))'
case EQUAL(__) then
    <<
<exp:LogEq>
    <%e1%>
    <%e2%>
</exp:LogEq>
>>
case NEQUAL(|ty = T_BOOL(__)|) then
    '((!<%e1%> && %<e2%>) || (>%e1%> && !%e2%))'
case NEQUAL(|ty = T_STRING(__)|) then
    '(!stringEqual(<%e1%>, <%e2%>))'
case NEQUAL(__) then
    <<
<exp:LogNeq>
    <%e1%>
"Helper to daeExpRelation."

match exp
case rel as RELATION(__) then
  match context
  case SIMULATION(genDiscrete=false) then
    match rel.optionExpisASUB
    case NONE() then
      let e1 = daeExpXml(rel.exp1, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
      let e2 = daeExpXml(rel.exp2, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
      let res =
        tempDeclXml("modelica_boolean", &varDecls /*BUFC*/)
      match rel.operator
      case LESS(__) then
        let &preExp +=
        <<
          <exp:LogLt>
          %e1
          %e2
        </exp:LogLt><%n>
      >>
      res
    case LESSEQ(__) then
      let &preExp +=
      <<
        <exp:LogLeq>
        %e1
        %e2
      </exp:LogLeq><%n>
    >>
    res
    case GREATER(__) then
      let &preExp +=
      <<
        <exp:LogGt>
        %e1
        %e2
      </exp:LogGt><%n>
    >>
    res
    case GREATEREQ(__) then
      let &preExp +=
      <<
        <exp:LogGeq>
        %e1
        %e2
      </exp:LogGeq><%n>
    >>
    res
  end match
  case SIMULATION(genDiscrete=true) then
    match rel.optionExpisASUB
    case NONE() then
      let e1 = daeExpXml(rel.exp1, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
      let e2 = daeExpXml(rel.exp2, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
      let res =
        tempDeclXml("modelica_boolean", &varDecls /*BUFC*/)
      match rel.operator
      case LESS(__) then
        let &preExp +=
        <<
          <exp:LogLt>
          %e1
          %e2
        </exp:LogLt><%n>
      >>
      res
    case LESSEQ(__) then
      let &preExp +=
      <<
        <exp:LogLeq>
        %e1
        %e2
      </exp:LogLeq><%n>
    >>
    res
    case GREATER(__) then
      let &preExp +=
      <<
        <exp:LogGt>
        %e1
        %e2
      </exp:LogGt><%n>
    >>
    res
    case GREATEREQ(__) then
      let &preExp +=
      <<
        <exp:LogGeq>
        %e1
        %e2
      </exp:LogGeq><%n>
    >>
    res
  end match
end match

case SOME((exp,i,j)) then
    let e1 = daeExpXml(rel.exp1, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
    let e2 = daeExpXml(rel.exp2, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
    let res =
        tempDeclXml("modelica_boolean", &varDecls /*BUFC*/)
        / let e3 = daeExp(createArray(i), context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
        / let iterator = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)

match rel.operator
    case LESS(__) then
        let &preExp +=
        <<
            <exp:LogLt>
                <%e1%>
            <%e2%>
            </exp:LogLt><%n%>
        >>
        res
    case LESSEQ(__) then
        let &preExp +=
        <<
            <exp:LogLeq>
                <%e1%>
            <%e2%>
            </exp:LogLeq> <%n%>
        >>
        res
    case GREATER(__) then
        let &preExp +=
        <<
            <exp:LogGt>
                <%e1%>
            <%e2%>
            </exp:LogGt><%n%>
        >>
        res
    case GREATEREQ(__) then
        let &preExp +=
        <<
            <exp:LogGeq>
                <%e1%>
            <%e2%>
            </exp:LogGeq><%n%>
        >>
        res
end match
end match
end match
end daeExpRelationSimXml;

template daeExpConstraintXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates XML for constraint"
::=
match exp
    case rel as RELATION(__) then
        match context
            case SIMULATION(genDiscrete=true) then
                match rel.optionExpisASUB
                    case NONE() then
                        let e1 = daeExpXml(rel.exp1, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
                        let e2 = daeExpXml(rel.exp2, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
                        let res =
                            tempDeclXml("modelica_boolean", &varDecls /*BUFC*/)
                            / let e3 = daeExp(createArray(i), context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
                            / let iterator = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)

match rel.operator
    case EQUAL(__) then
        <<
            <opt:ConstraintEqu>
                <%e1%>
            <%e2%>
            </opt:ConstraintEqu> <%n%>
        >>
    case LESSEQ(__) then
        <<
            <opt:ConstraintLeq>
                <%e1%>
            <%e2%>
            </opt:ConstraintLeq> <%n%>
        >>
    case GREATEREQ(__) then
        <<
            <opt:ConstraintGeq>
                <%e1%>
            <%e2%>
            </opt:ConstraintGeq><%n%>
        >>
    else
        "The XML schema does only support =, >=, <= operators for constraints"
        >>
        end match
end match
end daeExpRelationSimXml;

template daeExpIfXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates code for an if expression."
::=
match exp
    case IFEXP(__) then
        let condExp = daeExpXml(expCond, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
        let &preExpThen = buffer "" /*BUFP*/
        let eThen = daeExpXml(expThen, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
        let &preExpElse = buffer "" /*BUFP*/
        let eElse = daeExpXml(expElse, context, &preExp /*BUFC*/, &varDecls /*BUFC*/)
        let shortIfExp = if preExpThen then "" else if preExpElse then "" else if isArrayType(typeof(exp)) then "" else "x" (if shortIfExp
        then
            // Safe to do if eThen and eElse don't emit pre-expressions
            <<
                <fun:If>
                    <fun:Condition>
                        <%condExp%>
                    </fun:Condition>
                    <fun:Statements>
                        <%eThen%>
                    </fun:Statements>
                </fun:If>
            >>
            else
                "The XML schema does only support =, >=, <= operators for constraints"
                >>
                end match
            end match
        end match
end if
end daeExpRelationSimXml;

let res =
    tempDeclXml("modelica_boolean", &varDecls /*BUFC*/)
    match rel.operator
    case EQUAL(__) then
        <<
            <opt:ConstraintEq>
                <%e1%>
            <%e2%>
            </opt:ConstraintEq> <%n%>
        >>
    case LESSEQ(__) then
        <<
            <opt:ConstraintLeq>
                <%e1%>
            <%e2%>
            </opt:ConstraintLeq> <%n%>
        >>
    case GREATEREQ(__) then
        <<
            <opt:ConstraintGeq>
                <%e1%>
            <%e2%>
            </opt:ConstraintGeq><%n%>
        >>
    else
        "The XML schema does only support =, >=, <= operators for constraints"
        >>
        end match
end if
end daeExpRelationSimXml;
let condVar =
  tempDeclXml("modelica_boolean", &varDecs /*BUFD*/)
let resVarType =
  expTypeFromExpArrayIfXml(expThen)
let resVar = tempDeclXml(resVarType, &varDecs /*BUFD*/)
let &preExp += <<
  resVar)
end daeExpIfXml;

template daeExpCallXml(Exp call, Context context, Text &preExp /*BUFP*/, Text &varDecs /*BUFP*/)
"Generates code for a function call."
::=
  match call
    // special builtins
    case CALL(path=IDENT(name="DIVISION"),
      expLst={e1, e2, DAE.SCONST(string=string)}) then
      let var1 = daeExpXml(e1, context, &preExp, &varDecs)
      let var2 = daeExpXml(e2, context, &preExp, &varDecs)
      let var3 =
        Util.escapeModelicaStringToCString(string)
        <<
          <exp:Div>
            <%var1%>
            <%var2%>
          </exp:Div>
        >>
    case CALL(attr=CALL_ATTR(ty=ty),
      path=IDENT(name="DIVISION_ARRAY_SCALAR"),
      expLst={e1, e2, e3 as SHARED_LITERAL(__)}) then
      let type = match ty case
        T_ARRAY(ty=T_INTEGER(__)) then "integer_array"
        T_ARRAY(ty=T_ENUMERATION(__)) then "integer_array"
        else "real_array"
      end
      let var = tempDeclXml(type, &varDecs)
      let var1 = daeExpXml(e1, context, &preExp, &varDecs)
      let var2 = daeExpXml(e2, context, &preExp, &varDecs)
      let var3 = daeExpXml(e3, context, &preExp, &varDecs)
      let &preExp +=
        'division_alloc_<%type%>_scalar(&<%var1%>,
        <%var2%>, &<%var%>, <%var3%>);%
      testcallA'
    case exp as
      call(path=IDENT(name="DIVISION_ARRAY_SCALAR"))
      then error(sourceInfo(), 'Code generation does not support call(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="der"),
      expLst={exp}) then
      error(sourceInfo(), 'Code generation does not support der(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="pre"),
      expLst={arg}) then
      daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="$_start"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="edge"),
      expLst={arg as CREF(__)}) then
      <<
        <%crefXml(arg.componentRef)%>
      >>
    case CALL(path=IDENT(name="der"),
      expLst={exp}) then
      error(sourceInfo(), 'Code generation does not support der(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="pre"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="$_start"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="edge"),
      expLst={arg as CREF(__)}) then
      <<
        <%crefXml(arg.componentRef)%>
      >>
    case CALL(path=IDENT(name="der"),
      expLst={exp}) then
      error(sourceInfo(), 'Code generation does not support der(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="pre"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="$_start"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="edge"),
      expLst={arg as CREF(__)}) then
      <<
        <%crefXml(arg.componentRef)%>
      >>
    case CALL(path=IDENT(name="der"),
      expLst={exp}) then
      error(sourceInfo(), 'Code generation does not support der(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="pre"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="$_start"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="edge"),
      expLst={arg as CREF(__)}) then
      <<
        <%crefXml(arg.componentRef)%>
      >>
    case CALL(path=IDENT(name="der"),
      expLst={exp}) then
      error(sourceInfo(), 'Code generation does not support der(<%printExpStr(exp)%>)')
    case CALL(path=IDENT(name="pre"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="$_start"),
      expLst={arg}) then
daeExpCallPreXml(arg, context, preExp, varDecs)
    case CALL(path=IDENT(name="edge"),
      expLst={arg as CREF(__)}) then
      <<
        <%crefXml(arg.componentRef)%>
      >>
case CALL(path=IDENT(name="min"),
attr=CALL_ATTR(ty = T_REAL(_)),
expLst={e1,e2}) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
let var2 = daeExpXml(e2, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Min>
<%var1%>
<%var2%>
</exp:Min>
>>

case CALL(path=IDENT(name="min"),
expLst={e1,e2}) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
let var2 = daeExpXml(e2, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Min>
<%var1%>
<%var2%>
</exp:Min>
>>

case CALL(path=IDENT(name="abs"),
expLst={e1}, attr=CALL_ATTR(ty = TInteger(_))) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Abs>
<%var1%>
</exp:Abs>
>>

case CALL(path=IDENT(name="abs"),
expLst={e1}) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Abs>
<%var1%>
</exp:Abs>
>>

// sqrt
case CALL(path=IDENT(name="sqrt"),
expLst={e1}, attr as CALL_ATTR(_)) then
let retPre =
assertCommonXml(createAssertForSqrt(e1),createD
AEString("Model error: Argument of sqrt should
be \geq 0"), context, &varDecls, dummyInfo)
let argStr = daeExpXml(e1, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let &preExp += '<%retPre%>'
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Sqrt>
<%argStr%>
</exp:Sqrt>
>>

case CALL(path=IDENT(name="div"),
expLst={e1,e2}) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
let var2 = daeExpXml(e2, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'.quot testcallC'

<<
<exp:Div>%var1%<%var2%></exp:Div>
>>

case CALL(path=IDENT(name="mod"),
expLst={e1,e2}, attr=CALL_ATTR(ty = ty)) then
let var1 = daeExpXml(e1, context, &preExp, &varDecls)
let var2 = daeExpXml(e2, context, &preExp, &varDecls)
'\sum_{%ty_str%}(<%arr%>)'

<<
<exp:Mod>
<%var1%><%var2%></exp:Mod>
>>

case CALL(path=IDENT(name="max"),
attr=CALL_ATTR(ty = ty), expLst={array}) then
let expVar = daeExpXml(array, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let &preExp += '<%expVar%>'
'\sum_{%ty_str%}(<%arr%>)'

let arr_tp_str = '<%expTypeArrayXml(ty)%>
let tvar =
tempDeclXml(expTypeModelicaXml(ty), &varDecls
*/BUFD*/)
let &preExp += '<%tvar%> =
max_<%arr_tp_str%>(&<%expVar%>);<%\n%>'
'\sum_{%ty_str%}(<%arr%>)'

case CALL(path=IDENT(name="min"),
attr=CALL_ATTR(ty = ty), expLst={array}) then
let expVar = daeExpXml(array, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let &preExp += '<%expVar%>'
'\sum_{%ty_str%}(<%arr%>)'

let arr_tp_str = '<%expTypeArrayXml(ty)%>
let tvar =
tempDeclXml(expTypeModelicaXml(ty), &varDecls
*/BUFD*/)
let &preExp += '<%tvar%> =
min_<%arr_tp_str%>(&<%expVar%>);<%\n%>'
'\sum_{%ty_str%}(<%arr%>)'

case CALL(path=IDENT(name="fill"),
expLst=val::dims, attr=CALL_ATTR(ty = ty)) then
let valExp = daeExpXml(val, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let dimsExp = (dims |> dim =>
daeExpXml(dim, context, &preExp
*/BUFC*/, &varDecls */BUFD*/);
let &preExp += '<%expTypeArrayXml(ty)%>
let tvar =
tempDeclXml(ty_str, &varDecls
*/BUFD*/)
let &preExp +=
'fill_alloc_<%ty_str%>(&<%tvar%>, <%valExp%>,<%listLength(dims)%>, <%dimsExp%>);<%\n%>'
'\sum_{%ty_str%}(<%arr%>)'

case CALL(path=IDENT(name="vector")
then
error(sourceInfo(),'vector() call does not
have a C implementation <%printExpStr(call)%>
'

case CALL(path=IDENT(name="cat")
then
let dim_exp = daeExpXml(dim, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let arrays_exp = (arrays |> array =>
daeExpXml(array, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let &preExp += '</exp:Cat>'
'\sum_{%ty_str%}(<%arr%>)'

let ty_str = '<%expTypeArrayXml(ty)%>
let tvar =
tempDeclXml(ty_str, &varDecls
*/BUFD*/)
let &preExp += '</exp:Cat>'
'\sum_{%ty_str%}(<%arr%>)'

case call as CALL(path=IDENT(name="vector")
then
elementError(sourceInfo(),'vector() call does not
have a C implementation <%printExpStr(call)%>
'

case CALL(path=IDENT(name="cat")
then
let dim_exp = daeExpXml(dim, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let arrays_exp = (arrays |> array =>
daeExpXml(array, context, &preExp
*/BUFC*/, &varDecls */BUFD*/)
let &preExp += '</exp:Cat>'
'\sum_{%ty_str%}(<%arr%>)'

let ty_str = '<%expTypeArrayXml(ty)%>
let tvar =
tempDeclXml(ty_str, &varDecls
*/BUFD*/)
let &preExp += '</exp:Cat>'
'\sum_{%ty_str%}(<%arr%>)'
let &preExp += 'cat_alloc_<%ty_str%>(<%dim_exp%>, &<%tvar%>, <%listLength(arrays)%>, &<%arrays_exp%>);<%n%>'

\nwhere is cat2'

case CALL(path=IDENT(name="promote"), expList=(A, n)) then
  let var1 = daeExpXml(A, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let var2 = daeExpXml(n, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let arr tp_str = '<%expTypeFromExpArrayXml(A)%>'
  let tvar = tempDeclXml(arr tp_str, &varDecls /*BUFD*/)
  let &preExp += 'promote_alloc_<%arr tp_str%>(&<%var1%>, <%var2%>, &<%tvar%>);<%n%>'

case CALL(path=IDENT(name="transpose"), expList={A}) then
  let var1 = daeExpXml(A, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let arr tp_str = '<%expTypeFromExpArrayXml(A)%>'
  let tvar = tempDeclXml(arr tp_str, &varDecls /*BUFD*/)
  let &preExp += 'transpose_alloc_<%arr tp_str%>(&<%var1%>, &<%tvar%>);<%n%>'

case CALL(path=IDENT(name="cross"), expList={v1, v2}) then
  let var1 = daeExpXml(v1, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let var2 = daeExpXml(v2, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let arr tp_str = '<%expTypeFromExpArrayXml(v1)%>'
  let tvar = tempDeclXml(arr tp_str, &varDecs /*BUFD*/)
  let &preExp += 'cross_alloc_<%arr tp_str%>(&<%var1%>, &<%var2%>, &<%tvar%>);<%n%>'

let tvar = tempDeclXml("modelica_string", &varDecs /*BUFD*/)
let sExp = daeExpXml(s, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
let formatExp = daeExpXml(format, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
let &preExp += '<%tvar%> = modelica_string_to_modelica_string_format(<%sExp%>, <%formatExp%>);<%n%>'

case CALL(path=IDENT(name="String"), expList={s, minlen, leftjust}) then
  let tvar = tempDeclXml("modelica_string", &varDecs /*BUFD*/)
  let sExp = daeExpXml(s, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let minlenExp = daeExpXml(minlen, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let leftjustExp = daeExpXml(leftjust, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let &preExp += '<%tvar%> = modelica_string(<%sExp%>, <%minlenExp%>, <%leftjustExp%>);<%n%>'

case CALL(path=IDENT(name="String"), expList={s, minlen, leftjust, signdig}) then
  let tvar = tempDeclXml("modelica_string", &varDecs /*BUFD*/)
  let sExp = daeExpXml(s, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let minlenExp = daeExpXml(minlen, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let leftjustExp = daeExpXml(leftjust, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let signdigExp = daeExpXml(signdig, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let &preExp += '<%tvar%> = modelica_real_to_modelica_string(<%sExp%>, <%minlenExp%>, <%leftjustExp%>, <%signdigExp%>);<%n%>'

case CALL(path=IDENT(name="delay"), expList={ICONST(integer=index), e, d, delayMax}) then
  let tvar = tempDeclXml("modelica_real", &varDecs /*BUFD*/)
  let var1 = daeExpXml(e, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let var2 = daeExpXml(d, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let var3 = daeExpXml(delayMax, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let &preExp += 'delayImplXml(<%index%>, <%var1%>, time, <%var2%>, <%var3%>);<%n%>'

let tvar = tempDeclXml("modelica_string", &varDecs /*BUFD*/)

let var1 = daeExpXml(e1, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
let var2 = daeExpXml(e2, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
let tvar = tempDeclXml("modelica_string", &varDecs /*BUFD*/)
let &preExp += 'modelica_rem_<%typeStr%>(<%var1%>,<%var2%>);<%n%>'

case CALL(path=IDENT(name="integer"), expList={toBeCasted}) then
  let castedVar = daeExpXml(toBeCasted, context, &preExp /*BUFC*/, &varDecs /*BUFD*/)
  let &preExp += '<%castedVar%>;'
case CALL(path=IDENT(name="Integer"),
    expList={toBeCasted}) then
    let castedVar = daeExpXml(toBeCasted,
        context, &preExp /*BUFC*/, &varDecls /*BUFD*/)
    '<%castedVar%>'

case CALL(path=IDENT(name="clock"),
    expList={}) then
    'mmc_clock()'

case CALL(path=IDENT(name="noEvent"),
    expList={e1}) then
    daeExpXml(e1, context, &preExp, &varDecls)

case CALL(path=IDENT(name="anyString"),
    expList={e1}) then
    '<%daeExpXml(e1, context, &preExp, &varDecls)%>'

case CALL(path=IDENT(name="mmc_get_field"),
    expList={s1, ICONST(integer=i)}) then
    let tvar = tempDeclXml("modelica_metatype",
            &varDecls /*BUFD*/)
    let expPart = daeExpXml(s1, context,
            &preExp /*BUFC*/, &varDecls /*BUFD*/)
    let &preExp += '<%tvar%> = MMC_FETCH(MMC_OFFSET(MMC_UNTAGPTR(<%expPart%>),<%i%>));<%n%>'
    '<%tvar%>'

case CALL(path=IDENT(name = "mmc_unbox_record"),
    expList={s1}, attr=CALL_ATTR(ty=ty)) then
    <<
    "mmc_unbox_record" is not necessary
    >>

case exp as CALL(attr=attr as
    CALL_ATTR(tailCall=tail as TAIL(__))) then
    let res = <<
    /* Tail recursive call <%printExpStr(exp)%>
    */
    <%daeExpTailCallXml(expLst,tail.vars,context,&preExp,&varDecls)%>goto _tailrecursive; /* TODO: Make sure any eventual dead code below is never generated */

    let &preExp += res
    ""
    case exp as CALL(attr=attr as CALL_ATTR(__))
then
    let argStr = (expLst |> exp =>
        '<%daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/)%>' ;separator="\n")
    let builtinName ='<%dotPathXml(path)%>'
    let funName = '<%underscorePathXml(path)%>'
    let retType = if attr.builtin then (match attr.ty case T_NORETCALL(__) then "" else expTypeModelicaXml(attr.ty))
        else '<%funName%>
    let retVar = match attr.ty case T_NORETCALL(__) then "" else tempDeclXml(retType, &varDecls)
    match exp // no return calls case
    CALL(attr=CALL_ATTR(ty=T_NORETCALL(__))) then
        /* NORETCALL */
    // non tuple calls (single return value)
    case CALL(attr=CALL_ATTR(tuple_=false))
    then if attr.builtin then
        <<
        <exp:<%builtinName%>>
        <%argStr%>
        </exp:<%builtinName%>>
        >>
    else
        <<
        <exp:FunctionCall>
        <exp:Name>
        <%FunName%>
        </exp:Name>
        <exp:Arguments>
        <%argStr%>
        </exp:Arguments>
        </exp:FunctionCall>
        >>
        // tuple calls (multiple return values)
    else
        <<
        <exp:FunctionCall>
        <exp:Name>
        <%FunName%>
        </exp:Name>
        <exp:Arguments>
        <%argStr%>
        </exp:Arguments>
        </exp:FunctionCall>
        >>
end daeExpCallXml;

template daeExpTailCallXml(list<DAE.Exp> es, list<String> vs, Context context, Text &preExp, Text &varDecls)
::=
    match es
    case e::erest then
        match vs
        case v::vrest then
            let exp = daeExpXml(e,context,&preExp,&varDecls)
            match e
            case CREF(componentRef = cr, ty = T_FUNCTION_REFERENCE_VAR(__)) then
                adrpo: ignore _x = _x!
                if stringEq(v, crefStrXml(cr))
                    then '<%daeExpTailCallXml(erest, vrest, context, &preExp, &varDecls)%>'
                else '_<%v%> = <%exp%>;<%n%><%daeExpTailCallXml(erest, vrest, context, &preExp, &varDecls)%>'
            case _ then
                '_<%v%> = <%exp%>;<%n%><%daeExpTailCallXml(erest, vrest, context, &preExp, &varDecls)%>'
            end daeExpTailCallXml;
    end daeExpTailCallXml;
    template daeExpCallBuiltinPrefixXml(Boolean builtin)
    "Helper to daeExpCallXml."
    ::=
    match builtin
    case true  then ""
    case false then "_"
end daeExpCallBuiltinPrefixXml;

template daeExpArrayXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates code for an array expression."
::=
    match exp
    case ARRAY(_) then
        let params = (array |) e =>
        if attr.builtin then
            <<
            <exp:<%builtinName%>>
            <%argStr%>
            </exp:<%builtinName%>>
            >>
        else
            <<
            <exp:FunctionCall>
            <exp:Name>
            <%FunName%>
            </exp:Name>
            <exp:Arguments>
            <%argStr%>
            </exp:Arguments>
            </exp:FunctionCall>
            >>
        // tuple calls (multiple return values)
        else
            <<
            <exp:FunctionCall>
            <exp:Name>
            <%FunName%>
            </exp:Name>
            <exp:Arguments>
            <%argStr%>
            </exp:Arguments>
            </exp:FunctionCall>
            >>
end daeExpArrayXml;
case ASUB(exp=ASUB(____)) then
  error(sourceInfo(), 'Nested array subscripting *should* have been handled by the routine creating the asub, but for some reason it was not: <%printExpStr(exp)%>
}

// Faster asub: Do not construct a whole new array just to access one subscript

case ASUB(exp=ASUB(__)) then
  error(sourceInfo(), 'Nested array subscripting *should* have been handled by the routine creating the asub, but for some reason it was not: <%printExpStr(exp)%>
}

template daeExpASubIndexXml(Exp exp, Context context, Text &preExp, Text &varDecls)
::= match exp
  case ICONST(__) then incrementInt(integer, -1)
  case ENUM_LITERAL(__) then incrementInt(index, -1)
  else daeExpXml(exp, context, &preExp, &varDecls)
end daeExpASubIndexXml;

template daeExpCallPreXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates code for an asub of a cref, which becomes cref + offset."
::= match exp
case cr as CREF(____) then
  <<<exp:Pre>
    <%crefXml(cr.componentRef)%>
  </exp:Pre> >>
case ASUB(exp = cr as CREF(____), sub = {sub_exp}) then
  "case ASUB(exp = cr as CREF(____), sub = {sub_exp}) is not yet implemented"
else
  error(sourceInfo(), 'Code generation does not support pre(<%printExpStr(exp)%>)
end daeExpCallPreXml;

template daeExpSizeXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates XML code for a size expression."
::= match exp
case SIZE(exp=CREF(____), sz=SOME(dim)) then
  let expPart = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/
  let dimPart = daeExpXml(dim, context, &preExp /*BUFC*/, &varDecls /*BUFD*/
  <<
    <%expPart%>
    <%dimPart%>
  >>
else
  "size(X) not implemented"
end daeExpSizeXml;

template daeExpBoxXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates XML code for a match expression box."
::= match exp
case exp as BOX(____) then
  let res = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/
  <<
    <%res%>
  >>
end daeExpBoxXml;

template daeExpUnboxXml(Exp exp, Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates XML code for a match expression unbox."
::= match exp
case exp as BOX(____) then
  let res = daeExpXml(exp, context, &preExp /*BUFC*/, &varDecls /*BUFD*/
  <<
    <%res%>
  >>
end daeExpBoxXml;
match exp
case exp as UNBOX(__) then
  let res =
  daeExpXml(exp.exp,context,&preExp,&varDecls)
  <>
  <<res%>
end daeExpUnboxXml;

template daeExpSharedLiteralXml(Exp exp,
Context context, Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Generates code for a match expression."
::=
match exp case exp as SHARED_LITERAL(__) then
''  // alachew changed from
_OMC_LIT<%exp.index%>
end daeExpSharedLiteralXml;

template arrayScalarRhsXml(Type ty, list<Exp>
subs, String arrName, Context context,
Text &preExp /*BUFP*/, Text &varDecls /*BUFP*/)
"Helper to daeExpAsub."
::=
let arrayType = expTypeArrayXml(ty)
let dimsLenStr = listLength(subs)
let dimsValuesStr = (subs |> exp =>
daieExpXml(exp, context, &preExp /*BUFC*/,
&varDecls /*BUFD*/);
;separator=', ')
match arrayType
  case "metatype_array" then
'arrayGet(<%arrName%>,<%dimsValuesStr%>
)*arrayScalarRhs*'/
else
  << wrong LHS
  <exp:ArraySubscripts>
  <exp:IndexExpression>
  <%dimsValuesStr%>
  <exp:IndexExpression>
  </exp:ArraySubscripts>
  </exp:QualifiedNamepart>
  </exp:QualifiedName>
end arrayScalarRhsXml;

template outDeclXml(String ty, Text &varDecls
/*BUFP*/)
"Declares a temporary variable in varDecls and
returns the name."
::=
let newVar = 'out'
let &varDecls += '<%ty%> <%newVar%>;<%'n%>'
newVar
end outDeclXml;

template tempDeclXml(String ty, Text &varDecls
/*BUFP*/)
"Declares a temporary variable in varDecls and
returns the name."
::=
let newVar = 'out'
let &varDecls += '<%ty%> <%newVar%>;<%'n%>'
newVar
end tempDeclXml;

template varTypeXml(Variable var)
"Generates type for a variable."
::=
match var
  case var as VARIABLE(__) then
    if instDims then
      expTypeArrayXml(var.ty)
    else
      expTypeArrayIfXml(var.ty)
    end varTypeXml;
end tempDeclConstXml(String ty, String
val, Text &varDecls /*BUFP*/)
"Declares a temporary variable in varDecls and
returns the name."
::=
let newVar = 'tmp<%System.tmpTick()%>'
let &varDecls += '<%ty%> <%newVar%> =<%val%>;<%'n%>'
newVar
end tempDeclConstXml;

template expTypeRWXml(DAE.Type type)
"Helper to writeOutVarRecordMembers." 
::=
match type
  case T_INTEGER(__) then "TYPE_DESC_INT"
case T_REAL(__) then "TYPE_DESC_REAL"
case T_STRING(__) then "TYPE_DESC_STRING"
case T_BOOLEAN(__) then "TYPE_DESC_BOOL"
case T_ENUMERATION(__) then "TYPE_DESC_ENUM"
case T_ARRAY(__) then "<%expTypeRWXml(ty)%>_ARRAY"
case T_COMPLEX(complexClassType=EXTERNAL_OBJ(__)) then "complex"
case T_COMPLEX(__) then
' <%underscorePathXml(ClassInf.getStateName(complexClassType))%> '
end expTypeRWXml;

template expTypeShortXml(DAE.Type type)
"Generate type helper."
::=
match type
  case T_INTEGER(__) then "integer"
case T_REAL(__) then "real"
case T_STRING(__) then if
acceptMetaModelicaGrammar() then "string"
else "string"
case T_BOOLEAN(__) then "boolean"
case T_ENUMERATION(__) then "integer"
case T_ARRAY(__) then
expTypeShortXml(ty)
case T_COMPLEX(complexClassType=EXTERNAL_OBJ(__)) then "complex"
case T_COMPLEX(__) then
' <%underscorePathXml(ClassInf.getStateName(complexClassType))%> '
end expTypeShortXml;

case T_METATYPE(__) case T_METABOXED(__) then "metatype"
case T_FUNCTION_REFERENCE_VAR(__) then "fnptr"
case T_UNKNOWN(__) then "complex" /* TODO: Don't do this to me! */
case T_ANYTYPE(__) then "complex" /* TODO: Don't do this to me! */
else error(sourceInfo(),'expTypeShortXml:<%typeString(type)%>');</template>

template mmcVarTypeXml(Variable var)
::=
match var
case VARIABLE(__) then 'modelica_<%mmcTypeShortXml(ty)%>'
case FUNCTION_PTR(__) then 'modelica_fnptr'
end mmcVarTypeXml;

template mmcTypeShortXml(DAE.Type type)
::=
match type
case T_INTEGER(__) then "integer"
case T_REAL(__) then "real"
case T_STRING(__) then "string"
case T_BOOL(__) then "integer"
case T_ENUMERATION(__) then "integer"
case T_ARRAY(__) then "array"
case T_METATYPE(__) case T_METABOXED(__) then "metatype"
case T_FUNCTION_REFERENCE_VAR(__) then "fnptr"
else "mmcTypeShort:ERROR"
end mmcTypeShortXml;

template expTypeXml(DAE.Type ty, Boolean array)
"Generate type helper."
::=
match array
case true then expTypeArrayXml(ty)
case false then expTypeModelicaXml(ty)
end expTypeXml;

template expTypeModelicaXml(DAE.Type ty)
"Generate type helper."
::=
expTypeFlagXml(ty, 2)
end expTypeModelicaXml;

template expTypeArrayXml(DAE.Type ty)
"Generate type helper."
::=
expTypeFlagXml(ty, 3)
end expTypeArrayXml;

template expTypeArrayIfXml(DAE.Type ty)
"Generate type helper."
::=
expTypeFlagXml(ty, 4)
end expTypeArrayIfXml;

template expTypeFlagXml(DAE.Type ty, Integer flag)
"Generate type helper."
::=
match flag
case 1 then
end expTypeFlagXml;

template expTypeFromExpModelicaXml(Exp exp)
"Generate type helper."
::=
end expTypeFromExpModelicaXml;

template expTypeFromExpArrayXml(Exp exp)
"Generate type helper."
::=
end expTypeFromExpArrayXml;

template expTypeFromExpArrayIfXml(Exp exp)
"Generate type helper."
::=
end expTypeFromExpArrayIfXml;

template expTypeFromExpShortXml(Exp exp)
"Generate type helper."
::=
end expTypeFromExpShortXml;

template expTypeFromExpFlagXml(Exp exp, Integer flag)
"Generate type helper."
::=
match flag
case 1 then
end expTypeFromExpFlagXml;
B.2 Template Based DAE Unparser Implementation Code

package DAEDumpTpl

import interface DAEDumpTV;
import AbsynDumpTpl;
import SCodeDumpTpl;

/************************************
*     SECTION: MAIN TEMPLATE FUNCTION
*************************************/
template dumpDAE(list<DAEDump.compWithSplitElements> fixedDaeList, DAEDump.functionList funLists)
::=
  let comp_str = (fixedDaeList |>
dae => dumpComp(dae)) ; separator="\n"
  let fun_str = match funLists case
(FUNCTION_LIST(__)) then dumpFunctions(fun)
if fun_str then
  <<%fun_str%><%\n%>
  <%comp_str%>
>>
else
  <<%comp_str%>
>>
end dumpDAE;

template dumpComp(DAEDump.compWithSplitElements fixedDae) :
::=
  match fixedDae case COMP_WITH_SPLIT(__) then dumpCompStream(splitElems)
  let cmt_str = dumpCommentOpt(comment) <<<%AbsynDumpTpl.dumpPathNoQual(path)%><%cmt_str%>
  <%dumpCompStream(spltElems)%>
<%AbsynDumpTpl.dumpPathNoQual(path)%>;
end dumpComp;

template dumpCompStream(DAEDump.splitElements elems)
::=
  match elems case SPLIT_ELEMENTS(__) then dumpCompStream(splitElems)
  let var_str = dumpVars(v)
  let ieq_str = dumpInitialEquations(ie, "initial equation")
  let ial_str = dumpInitialAlgorithms(ia, "initial algorithm")
  let eq_str = dumpEquations(e, "equation")
  let al_str = dumpAlgorithms(a, "algorithm")
  <<%var_str%>
  <kieq_str%>
  <%ial_str%>
  <seq_str%>
  <%al_str%>
>>
end dumpCompStream;

/************************************
*     SECTION: FUNCTION SECTION
*************************************/
template dumpFunctions(list<DAE.Function> funcs)
::=
  (funcs |> func => dumpFunction(func)) ; separator="\n\n"
end dumpFunctions;

template dumpFunction(DAE.Function function)
::=
  match function case
  FUNCTION(fns) then ''
  case FUNCTION(__) then
    let cnt_str = dumpCommentOpt(comment)
    "Automatically generated record constructor for %AbsynDumpTpl.dumpPathNoQual(path)%" %cnt_str%
    <%dumpFunctionDefinitions(fns)%>
    end dumpFunction;
end dumpFunction;

template dumpFunctionDefinitions(DAE.FunctionDefinition functions)
::=
  (functions |>
def => dumpFunctionDefinition(def)) ; separator="\n"
end dumpFunctionDefinitions;

template dumpFunctionDefinition(DAE.FunctionDefinition def)
::=
  match def case FUNCTION_DEF(__) then
    <<%dumpElements(body)%>
  case FUNCTION_EXT(__) then
    <<%dumpElements(body)%><%\n%><%\n%>
    <%dumpExternalDecl(externalDecl)%>
  case FUNCTION_DER_MAPPER(__) then ''
end dumpFunctionDefinition;

template dumpExternalDecl(ExternalDecl externalDecl)
::=
  match externalDecl case EXTERNALDECL(__) then
    let func_name_str = name
    let func_args_str = dumpExtArgs(args)
    "%Automatically generated externaldec for %AbsynDumpTpl.dumpPathNoQual(path)%" %func_name_str% %func_args_str%
    "Automatically generated externaldec for %AbsynDumpTpl.dumpPathNoQual(path)%" %func_name_str% %func_args_str%
    "Automatically generated externaldec for %AbsynDumpTpl.dumpPathNoQual(path)%" %func_name_str% %func_args_str%
    "Automatically generated externaldec for %AbsynDumpTpl.dumpPathNoQual(path)%" %func_name_str% %func_args_str%
end dumpExternalDecl;

template dumpExternDec(ExternalDec externalDec)
::=
  match externalDec case EXTERNALDEC(__) then
    let func_name_str = name
    let func_args_str = dumpExtArgs(ARGS)
end dumpExternDec;
let func_str = if func_name_str then ' <%func_name_str%>(%func_args_str%)'

let ext_output_str = if ext_output_str then 'external "%lang_str%" <%output_str%>'

end dumpExternalDecl;

template dumpExtArgs(list<ExtArg> args) ::= (args |> arg => dumpExtArg(arg));separator=""," end dumpExtArgs;

template dumpExtArg(DAE.ExtArg arg) ::= match arg case EXTARG(__) then dumpCref(componentRef)
case EXTARGEXP(__) then dumpExp(exp)
case EXTARGSIZE(__) then 'size(<%dumpCref(componentRef)>,%dumpExp(exp))'
end dumpExtArg;

template dumpRecordInputVarStr(Type type_) ::= match type_ case T_COMPLEX(__) then '
<%dumpRecordVars(varLst)%>
case T_FUNCTION(__) then '
<%dumpRecordInputVarStr(funcResultType)%>
end dumpRecordInputVarStr;

template dumpRecordVars(list<Var> varLst) ::= (varLst |> v => dumpRecordVar(v);separator="\n")
end dumpRecordVars;

template dumpRecordVar(DAE.Var v) ::= match v case TYPES_VAR(attributes = DAE.ATTR(visibility=SCode.PROTECTED())) then <<
default protected <%varName%>
<%dumpRecordVarBinding(binding)%>
>> case TYPES_VAR(visibility=SCode .CONST()) then <<
default constant <%varName%>
<%dumpRecordVarBinding(binding)%>
>> case TYPES_VAR(__) then <<
default <%varName%>
<%dumpRecordVarBinding(binding)%>
>>
end dumpRecordVar;

template dumpRecordVarBinding(Binding binding) ::= match binding case UNBOUND(__) then ''
case EQBOUND(__) then ' = <%dumpExp(exp)%>'
case VALBOUND(__) then 'value bound*****
check what to display''
end dumpRecordVarBinding;

template dumpElements(list<Element> dAElist) ::= (dAElist |> lst => dumpElement(lst);separator="\n")
end dumpElements;

template dumpElement(DAE.Element lst) ::= match lst case VAR(__) then dumpVar(lst,true)
case INITIALALGORITHM(__) then dumpAlgorithmElement(algorithm_,"initial algorithm")
case ALGORITHM(__) then dumpAlgorithmElement(algorithm_,"algorithm")
else 'Element not found'
end dumpElement;

template dumpAlgorithmElement(Algorithm algorithm_, String label) ::= match algorithm_ case ALGORITHM_STMTS(__) then <<
<%label%>
<%dumpStatements(statementLst)%>
>>
end dumpAlgorithmElement;

template dumpVar(DAE.Element lst, Boolean printTypeDimension) ::= match lst case VAR(__) then let final = match variableAttributesOption case SOME(VariableAttributes) then dumpFinalPrefix(VariableAttributes)
let varVisibility = dumpVarVisibility(protection)
let varParallelism = dumpVarParallelism(parallelism)
let varKind = dumpVarKind(kind)
let varDirection = dumpVarDirection(direction)
let varType = dumpVarType(type)
let dim_str = if printTypeDimension then dumpTypeDimensions(dims)
let varName = dumpCref(componentRef)
let bindingExp = match binding case SOME(exp) then dumpExp(exp)
let varAttr = match variableAttributesOption case SOME(VariableAttributes) then dumpVariableAttributesStr(VariableAttributes)
let cmnt_str = dumpCommentOpt(absynCommentOption)
if bindingExp then <<
<%final%><%varVisibility%><%varParallelism%><%varKind%>
<%varDirection%><%varType%>
<%dim_str%>
<%varName%><%bindingExp%>
<%varAttr%>
<%cmnt_str%>
>>
end dumpVar;
template dumpFinalPrefix(DAE.VariableAttributes varAttr) ::= 
match varAttr 
  case VAR_ATTR_REAL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_INT(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_BOOL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_STRING(finalPrefix=SOME(true)) then ' final' 
end dumpFinalPrefix;

template dumpVarVisibility(VarVisibility protection) ::= 
match protection 
  case PROTECTED(__) then ' protected' 
end dumpVarVisibility;

template dumpVarParallelism(VarParallelism parallelism) ::= 
match parallelism 
  case PARGLOBAL(__) then 'parglobal' 
  case PARLOCAL(__) then 'parlocal' 
end dumpVarParallelism;

template dumpVarKind(VarKind kind) ::= 
match kind 
  case CONST(__) then 'constant' 
  case PARAM(__) then 'parameter' 
  case DISCRETE(__) then 'discrete' 
end dumpVarKind;

template dumpVarDirection(VarDirection direction) ::= 
match direction 
  case INPUT(__) then 'input' 
  case OUTPUT(__) then 'output' 
end dumpVarDirection;

template dumpVarType(Type tv) ::= 
match tv 
  case T_INTEGER(varLst = {}) then 'Integer' 
  case T_REAL(varLst = {}) then 'Real' 
  case T_STRING(varLst = {}) then 'String' 
  case T_BOOL(varLst = {}) then 'Boolean' 
  case T_ENUMERATION(__) then 'enumeration(<%dumpEnumVars(literalVarLst)%>)' 
  case T_OF(__) then 'of' 
end dumpVarType;

template dumpFinalPrefix(DAE.VariableAttributes varAttr) ::= 
match varAttr 
  case VAR_ATTR_REAL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_INT(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_BOOL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_STRING(finalPrefix=SOME(true)) then ' final' 
end dumpFinalPrefix;

template dumpVarVisibility(VarVisibility protection) ::= 
match protection 
  case PROTECTED(__) then ' protected' 
end dumpVarVisibility;

template dumpVarParallelism(VarParallelism parallelism) ::= 
match parallelism 
  case PARGLOBAL(__) then 'parglobal' 
  case PARLOCAL(__) then 'parlocal' 
end dumpVarParallelism;

template dumpVarKind(VarKind kind) ::= 
match kind 
  case CONST(__) then 'constant' 
  case PARAM(__) then 'parameter' 
  case DISCRETE(__) then 'discrete' 
end dumpVarKind;

template dumpVarDirection(VarDirection direction) ::= 
match direction 
  case INPUT(__) then 'input' 
  case OUTPUT(__) then 'output' 
end dumpVarDirection;

template dumpVarType(Type tv) ::= 
match tv 
  case T_INTEGER(varLst = {}) then 'Integer' 
  case T_REAL(varLst = {}) then 'Real' 
  case T_STRING(varLst = {}) then 'String' 
  case T_BOOL(varLst = {}) then 'Boolean' 
  case T_ENUMERATION(__) then 'enumeration(<%dumpEnumVars(literalVarLst)%>)' 
  case T_OF(__) then 'of' 
end dumpVarType;

template dumpFinalPrefix(DAE.VariableAttributes varAttr) ::= 
match varAttr 
  case VAR_ATTR_REAL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_INT(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_BOOL(finalPrefix=SOME(true)) then ' final' 
  case VAR_ATTR_STRING(finalPrefix=SOME(true)) then ' final' 
end dumpFinalPrefix;

template dumpVarVisibility(VarVisibility protection) ::= 
match protection 
  case PROTECTED(__) then ' protected' 
end dumpVarVisibility;

template dumpVarParallelism(VarParallelism parallelism) ::= 
match parallelism 
  case PARGLOBAL(__) then 'parglobal' 
  case PARLOCAL(__) then 'parlocal' 
end dumpVarParallelism;

template dumpVarKind(VarKind kind) ::= 
match kind 
  case CONST(__) then 'constant' 
  case PARAM(__) then 'parameter' 
  case DISCRETE(__) then 'discrete' 
end dumpVarKind;

template dumpVarDirection(VarDirection direction) ::= 
match direction 
  case INPUT(__) then 'input' 
  case OUTPUT(__) then 'output' 
end dumpVarDirection;

template dumpVarType(Type tv) ::= 
match tv 
  case T_INTEGER(varLst = {}) then 'Integer' 
  case T_REAL(varLst = {}) then 'Real' 
  case T_STRING(varLst = {}) then 'String' 
  case T_BOOL(varLst = {}) then 'Boolean' 
  case T_ENUMERATION(__) then 'enumeration(<%dumpEnumVars(literalVarLst)%>)' 
  case T_OF(__) then 'of' 
end dumpVarType;
let uncertainty_str =
dumpUncertaintyStrs(uncertainOption)
let attrs_str = {quantity_str, unit_str,
displayunit_str, min_max_str, start_str,
fixed_str, nominal_str, stateSel_str,
uncertainty_str};separator="", "
if attrs_str then
<<
({attrs_str%})
>>
case VAR_ATTR_INT(__) then
let quantity_str =
dumpQuantityAttribute(quantity)
let start_str =
dumpInitialAttribute(initial_)
let fixed_str = dumpFixedAttribute(fixed)
let min_max_str = dumpMinMaxAttribute(min)
let attrs_str = {quantity_str,
min_max_str, start_str, fixed_str}
;separator="", "
if attrs_str then
<<
({attrs_str%})
>>
case VAR_ATTR_BOOL(__) then
let quantity_str =
dumpQuantityAttribute(quantity)
let start_str =
dumpInitialAttribute(initial_)
let fixed_str = dumpFixedAttribute(fixed)
let min_max_str = dumpMinMaxAttribute(min)
let attrs_str = {quantity_str,
min_max_str, start_str, fixed_str}
;separator="", "
if attrs_str then
<<
({attrs_str%})
>>
case VAR_ATTR_STRING(__) then
let quantity_str =
dumpQuantityAttribute(quantity)
let start_str =
dumpInitialAttribute(initial_)
let attrs_str = {quantity_str,
start_str}
if attrs_str then
<<
({attrs_str%})
>>
case VAR_ATTR_ENUMERATION(__) then
let quantity_str =
dumpQuantityAttribute(quantity)
let start_str =
dumpInitialAttribute(start)
let fixed_str = dumpFixedAttribute(fixed)
let min_max_str = dumpMinMaxAttribute(min)
let attrs_str = {quantity_str,
min_max_str, start_str, fixed_str}
;separator="", "
if attrs_str then
<<
({attrs_str%})
>>
end dumpVariableAttributesStr;

template dumpQuantityAttribute(Option<Exp> quantity)
::=
match quantity
case SOME(exp) then 'quantity = <dumpExp(exp)>'
end dumpQuantityAttribute;

template dumpUnitAttribute(Option<Exp> unit)
::=
match unit
case SOME(exp) then 'unit = <dumpExp(exp)>'
end dumpUnitAttribute;

template dumpDisplayUnitAttribute(Option<Exp> displayUnit)
::=
match displayUnit
case SOME(exp) then 'displayUnit = <dumpExp(exp)>'
end dumpDisplayUnitAttribute;

template dumpInitialAttribute(Option<Exp> initial_)
::=
match initial_
case SOME(exp) then 'start = <dumpExp(exp)>'
end dumpInitialAttribute;

template dumpFixedAttribute(Option<Exp> fixed)
::=
match fixed
case SOME(exp) then 'fixed = <dumpExp(exp)>'
end dumpFixedAttribute;

template dumpMinMaxAttribute(tuple<Option<Exp>, Option<Exp>> min)
::=
match min
case (SOME(exp1), SOME(exp2)) then 'min = <dumpExp(exp1)>, max = <dumpExp(exp2)>'
case (NONE(), SOME(exp2)) then 'max = <dumpExp(exp2)>'
case (SOME(exp1),NONE()) then 'min = <dumpExp(exp1)>'
end dumpMinMaxAttribute;

template dumpNominalAttribute(Option<Exp> nominal)
::=
match nominal
case SOME(exp) then 'nominal = <dumpExp(exp)>'
end dumpNominalAttribute;

template dumpStateSelectStrs(Option<StateSelect> stateS)
::=
match stateS
case SOME(StateSelect) then dumpStateSelectStr(StateSelect)
end dumpStateSelectStrs;

template dumpStateSelectStr(DAE.StateSelect stateS)
::=
match stateS
case NEVER(__) then 'StateSelect = StateSelect.never'
case AVOID(__) then 'StateSelect = StateSelect.avoid'
case DEFAULT(__) then 'StateSelect = StateSelect.default'
case PREFER(__) then 'StateSelect = StateSelect.prefer'
case ALWAYS(__) then 'StateSelect = StateSelect.always'
end dumpStateSelectStr;

template dumpUncertaintyStrs(Option<Uncertainty> uncertain)
::=

match uncertain
    case SOME(Uncertainty) then
dumpUncertaintyStr(Uncertainty)
edumpUncertaintyStrs;

template dumpUncertaintyStrs(DAE.Uncertainty uncertain)
::=
match uncertain
    case GIVEN(__) then 'StateSelect = Uncertainty.given'
    case SOUGHT(__) then 'StateSelect = Uncertainty.sought'
    case REFINE(__) then 'StateSelect = Uncertainty.refine'
edumpUncertaintyStrs;

template dumpCref(ComponentRef c)
::=
match c
    case CREF_QUAL(__) then
        <<
        <%ident%><%dumpSubscripts(subscriptLst)%>.<%dumpCref(componentRef)%>
        >>
    case CREF_IDENT(ident = "$DER") then
        <<
der(<%ident%><%dumpSubscripts(subscriptLst)%>)
        >>
    case CREF_IDENT(__) then
        <<
        <%ident%><%dumpSubscripts(subscriptLst)%>
        >>
end dumpCref;

template dumpInitialEquations(list<DAE.Element> ie,String label)
::=
if ie then
    <<
    <%label%>
    |
    ineq => dumpInitialEquation(ineq)
    ;separator="\n"
    >>
edumpInitialEquations;

template dumpInitialEquation(DAE.Element lst)
::=
match lst
    case INITIALDEFINE(__) then
        let lhs_str = dumpCref(componentRef)
        let rhs_str = dumpExp(exp)
        <<
        <%lhs_str%> = <%rhs_str%>
        >>
    case INITIAL_ARRAY_EQUATION(__) then
        let lhs_str = dumpExp(exp)
        let rhs_str = dumpExp(array)
        <<
        <%lhs_str%> = <%rhs_str%>
        >>
    case INITIAL_COMPLEX_EQUATION(__) then
        let lhs_str = dumpExp(lhs)
        let rhs_str = dumpExp(rhs)
        <<
        <%lhs_str%> = <%rhs_str%>
        >>
    case INITIAL_IF_EQUATION(__) then
        dumpInitialIfEquation(lst)
    case INITIALEQUATION(__) then
        let lhs_str = dumpExp(exp1)
        let rhs_str = dumpExp(exp2)
        <<
        <%lhs_str%> = <%rhs_str%>
        >>
edumpInitialEquation;

template dumpInitialIfEquation(DAE.Element lst)
::=
match lst
    case INITIAL_CONDITION(condition1 = if_cond :: elseif_conds, equations2 = if_branch :: elseif_branches) then
        let if_cond_str = dumpExp(if_cond)
        let if_branch_str = (if_branch |> e => dumpEquation(e) ;separator="\n")
        let elseif_str = dumpElseIfEquation(elseif_conds, elseif_branches)
        let else_str = if equations3 then
            <<
            else <%equations3 |> e => dumpEquation(e) ;separator="\n"
            >>
        else
            <<
            <%if_cond_str%> then <%if_branch_str%>
            <%elseif_str%>
            <%else_str%>
            end if;
        end if;
edumpInitialIfEquation;

/****** SECTION: INITIAL EQUATION SECTION ***********/
template dumpInitialEquations(list<DAE.Element> ie, String label)
::=
if ie then
    <<
    <%label%>
    
    | ineq => dumpInitialEquation(ineq)
    ;separator="\n"
    >>
edumpInitialEquations;

template dumpInitialIfEquation(DAE.Element lst)
::=
match lst
    case INITIAL_CONDITION(condition1 = if_cond :: elseif_conds, equations2 = if_branch :: elseif_branches) then
        let if_cond_str = dumpExp(if_cond)
        let if_branch_str = (if_branch |> e => dumpEquation(e) ;separator="\n")
        let elseif_str = dumpElseIfEquation(elseif_conds, elseif_branches)
        let else_str = if equations3 then
            <<
            else <%equations3 |> e => dumpEquation(e)
            ;separator="\n"
            >>
        else
            <<
            <%if_cond_str%> then <%if_branch_str%>
            <%elseif_str%>
            <%else_str%>
            end if;
        end if;
edumpInitialIfEquation;

/****** SECTION: EQUATION SECTION ***********/
template dumpEquations(list<DAE.Element> e, String label)
::=
if e then
    <<
    <%label%>
template dumpEquation(DAE.Element lst)
::=
match lst
  case EQUATION(__) then
    let lhs_str = dumpExp(exp)
    let rhs_str = dumpExp(scalar)
    <<
    <%lhs_str%> = <%rhs_str%;
  >>
  case EQUEQUATION(__) then
    let lhs_cref = dumpCref(cr1)
    let rhs_cref = dumpCref(cr2)
    <<
    <%lhs_cref%> = <%rhs_cref%;
  >>
  case ARRAY_EQUATION(__) then
    let lhs_str = dumpExp(exp)
    let rhs_str = dumpExp(array)
    <<
    <%lhs_str%> = <%rhs_str%;
  >>
  case COMPLEX_EQUATION(__) then
    let lhs_str = dumpExp(lhs)
    let rhs_str = dumpExp(rhs)
    <<
    <%lhs_str%> = <%rhs_str%;
  >>
  case DEFINE(__) then
    let lhs_str = dumpCref(componentRef)
    let rhs_str = dumpExp(exp)
    <<
    <%lhs_str%> = <%rhs_str%;
  >>
  case WHEN_EQUATION(__) then
    dumpWhenEquation(lst)
  case IF_EQUATION(__) then dumpIfEquation(lst)
  case ASSERT(__) then
    let cond_str = dumpExp(condition)
    let msg_str = dumpExp(message)
    <<
    assert(<%cond_str%>,<%msg_str%;
  >>
  case TERMINATE(__) then
    let msg_str = dumpExp(message)
    <<
    terminate(<%msg_str%;
  >>
  case REINIT(__) then
    let cref_str = dumpCref(componentRef)
    let exp_str = dumpExp(exp)
    <<
    reinit(<%cref_str%>,<%exp_str%;
  >>
  case NORETCALL(__) then 'NO RETURN CALL'
else 'UNKNOWN EQUATION TYPE'
end dumpEquation;

template dumpWhenEquation(DAE.Element lst)
::=
match lst
  case WHEN_EQUATION(__) then
    dumpWhenEquation(lst)
  case IF_EQUATION(__) then dumpIfEquation(lst)
  case ASSERT(__) then
    dumpWhenEquation(lst)
  case TERMINATE(__) then
dumpWhenEquation(lst)
  case REINIT(__) then
dumpWhenEquation(lst)
  case NORETCALL(__) then
dumpWhenEquation(lst)
else 'UNKNOWN EQUATION TYPE'
end dumpWhenEquation;

template dumpIfEquation(DAE.Element lst)
::=
match lst
  case IF_EQUATION(condition1 = if_cond ::
    elseif_conds, equations2 = if_branch ::
    elseif_branches) then
    let if_cond_str = dumpExp(if_cond)
    let if_branch_str = (if_branch |> e => dumpEquation(e) ;separator="\n")
    let elseif_str =
dumpElseIfEquation(elseif_conds, elseif_branches)
    let else_str = if equations3 then
      <<
      %else_str
    else
      <<
      elseif_str
    >>
    end if;
  >>
end dumpIfEquation;

template dumpElseIfEquation(list<Exp> condition1, list<list<DAE.Element>> equations)
::=
match condition1
  case cond :: rest_conds then
    match equations
      case branch :: rest_branches then
        let cond_str = dumpExp(cond)
        let branch_str = (branch |> e => dumpEquation(e) ;separator="\n")
        let rest_str =
dumpElseIfEquation(rest_conds, rest_branches)
        <<
        elseif_str
      else rest_str then
        <<
        %else_str
      >>
    end dumpElseIfEquation;
  else 'UNKNOWN ELSE IF EQUATION TYPE'
end dumpElseIfEquation;

/*****************************/
* SECTION: INITIAL ALGORITHM SECTION
******************************/

template dumpInitialAlgorithms(list<DAE.Element> ia, String label)
::=
match ia
  case WHEN_EQUATION(__) then
    dumpInitialAlgorithms(ia)
  case IF_EQUATION(__) then dumpInitialAlgorithms(ia)
  case DEFINE(__) then dumpInitialAlgorithms(ia)
  case ASSERT(__) then
dumpInitialAlgorithms(ia)
else 'UNKNOWN EQUATION TYPE'
end dumpInitialAlgorithms;
template dumpInitialAlgorithm(DAE.Element alg) ::= 
metha alg 
  case INITIALALGORITHM(_) then 
    <<
      %dumpAlgorithmStatement(algorithm_)
    >>
  end dumpInitialAlgorithm;

/*************************************
*     SECTION: ALGORITHM SECTION
*************************************

template dumpAlgorithms(list<DAE.Element> a, String label) ::= 
  if a then 
    <<
      %label
      <%a |> alg => dumpAlgorithm(alg) ;separator="\n"%>
    >>
  end dumpAlgorithms;

template dumpAlgorithm(DAE.Element alg) ::= 
  match alg 
    case ALGORITHM(_) then 
      <<
        %dumpAlgorithmStatement(algorithm_)
      >>
    end dumpAlgorithm;

template dumpAlgorithmStatement(Algorithm algorithm_) ::= 
  match algorithm_ 
    case ALGORITHM_STMTS(_)_ then 
      <<
        %dumpStatements(statementLst)
      >>
    end dumpAlgorithmStatement;

template dumpStatements(list<DAE.Statement> stmts) ::= 
  (stmts |> stmt => dumpStatement(stmt) ;separator="\n")
end dumpStatements;

template dumpStatement(DAE.Statement stmt) ::= 
  match stmt 
    case STMT_ASSIGN(_) then 
      let lhs_str = dumpExp(exp1)
      let rhs_str = dumpExp(exp)
      '<%lhs_str%> := <%rhs_str%>;'
    case STMT_TUPLE_ASSIGN(_) then 
      let lhs_str = (expExpLst |> e => dumpExp(e);separator="\n")
      let rhs_str = dumpExp(exp)
      <<
        <%lhs_str%> := <%rhs_str%;
      >>
    case STMT_AssIGN_ARR(_) then 
      let lhs_str = dumpCref(componentRef)
      let rhs_str = dumpExp(exp)
      <<
        <%lhs_str%> := <%rhs_str%;
      >>
    case STMT_IF(_) then 
      let if_cond_str = dumpExp(exp)
      let true_branch_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      let else_if_str = 
        dumpElseIfStatements(else_)
      <<
        if <%if_cond_str%> then 
          <%true_branch_str%>
        <%else_if_str%>
      >>
    case STMT_WHILE(_) then
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_WHEN(_) then
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_ASSERT(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_TERMINATE(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_REINIT(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    else errorMsg("DAEDump.dumpStatement: Unknown statement.")
  end dumpStatement;

template dumpTupleAssignStatement(DAE.Statement stmt) ::= 
  match stmt 
    case STMT_TUPLE_ASSIGN(_) then 
      let lhs_str = (expExpLst |> e => dumpExp(e);separator="\n")
      let rhs_str = dumpExp(exp)
      <<
        <%lhs_str%> := <%rhs_str%;
      >>
    end dumpTupleAssignStatement;

template dumpArrayAssignStatement(DAE.Statement stmt) ::= 
  match stmt 
    case STMT_ASSIGN_ARR(_) then 
      let lhs_str = dumpCref(componentRef)
      let rhs_str = dumpExp(exp)
      <<
        <%lhs_str%> := <%rhs_str%;
      >>
    end dumpArrayAssignStatement;

template dumpIfStatement(DAE.Statement stmt) ::= 
  match stmt 
    case STMT_IF(_) then 
      let if_cond_str = dumpExp(exp)
      let true_branch_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      let else_if_str = 
        dumpElseIfStatements(else_)
      <<
        if <%if_cond_str%> then 
          <%true_branch_str%>
        <%else_if_str%>
      >>
    end dumpIfStatement;

template dumpElseIfStatements(Else else_) ::= 
  match else_ 
    case ELSEIF(_) then 
      let elseif_cond_str = dumpExp(exp)
      let elseif_body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      let else_str = dumpElseIfStatements(else_)
      <<
        elseif <%elseif_cond_str%> then 
          elseif_body_str
        else 
      end if;
      >>
    case ELSE(__) then 
      let else_body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        else
        else_body_str
      >>
    end dumpElseIfStatements;

template dumpElseIfStatements(Else else_) ::= 
  match else_ 
    case ELSEIF(_) then 
      let elseif_cond_str = dumpExp(exp)
      let elseif_body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      let else_str = dumpElseIfStatements(else_)
      <<
        elseif <%elseif_cond_str%> then 
          elseif_body_str
        else 
      end if;
      >>
    case ELSE(__) then 
      let else_body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        else
        else_body_str
      >>
    end dumpElseIfStatements;

template dumpTerminateStatement(DAE.Statement stmt) ::= 
  match stmt 
    case STMT_TERMINATE(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_NORETCALL(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_RETURN(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    case STMT_BREAK(_) then 
      let cond_str = dumpExp(exp)
      let body_str = (statementLst |> e => dumpStatement(e) ;separator="\n")
      <<
        <%cond_str%>:
        <%body_str%>
      >>
    else errorMsg("DAEDump.dumpStatement: Unknown statement.")
  end dumpTerminateStatement;
::= match stmt
  case STMT_FOR(_) then
    let range_str = dumpExp(range)
    let alg_str = (statementList |> e =>
      dumpStatement(e) ;separator="\n")
    <<
      for <%iter%> in <%range_str%> loop
        <%alg_str%>
    end for;
  >> end dumpForStatement;

  template dumpWhileStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_WHILE(_) then
         let while_cond = dumpExp(exp)
         let body_str = (statementList |> e =>
           dumpStatement(e) ;separator="\n")
         <<
           while <%while_cond%> loop
             <%body_str%>
           end while;
       >> end dumpWhileStatement;

  template dumpWhenStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_WHEN(_) then
         let when_cond_str = dumpExp(exp)
         let body_str = (statementList |> e =>
           dumpStatement(e) ;separator="\n")
         let elsewhen_str = match stmt case
           STMT_WHEN(elseWhen = SOME(Statement)) then
             dumpStatement(Statement)
           else when elseWhen then
             '<%elsewhen_str%>'
         end when
       >> end dumpWhenStatement;

  template dumpAssertStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_ASSERT(_) then
         let assert_cond = dumpExp(cond)
         let assert_msg = dumpExp(msg)
         <<
           assert(<%assert_cond%>, <%assert_msg%>);
       >> end dumpAssertStatement;

  template dumpTerminateStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_TERMINATE(_) then
         let msg_str = dumpExp(msg)
         <<
           terminate(<%msg_str%>);
       >> end dumpTerminateStatement;

  template dumpReinitStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_REINIT(_) then
         let exp_str = dumpExp(var)
         let new_exp_str = dumpExp(value)
         <<
           reinit(<%exp_str%>, <%new_exp_str%>);
       >> end dumpReinitStatement;

  template dumpNoReturnCallStatement(DAE.Statement stmt)
    ::= match stmt
       case STMT_NORETCALL(_) then
         let exp_str = dumpExp(exp)
         <<
           <%exp_str%>;
       >> end dumpNoReturnCallStatement;

/****************************
*     SECTION: EXPRESSIONS
****************************/
template dumpExp(DAE.Exp exp)
  ::= ExpressionDumpTpl.dumpExp(exp, "")
edumpExp;

template dumpCommentOpt(Option<SCode.Comment> comment)
  ::= match comment case SOME(cmt) then
      dumpComment(cmt)
      end dumpCommentOpt;

template dumpComment(SCode.Comment comment)
  ::= match comment case COMMENT(__) then
      let cmt_str = dumpCommentStr(comment)
      '<%cmt_str%>'
    case CLASS_COMMENT(__) then
      let cmt_str = dumpCommentOpt(comment)
      '<%cmt_str%>'
    end dumpComment;

template dumpCommentStr(Option<String> comment)
  ::= match comment case SOME(cmt) then
      '<%cmt%>'
  end dumpCommentStr;

template dumpPathLastIndent(Absyn.Path path)
  ::= match path case FULLYQUALIFIED(__) then
      dumpPathLastIndent(path)
    case QUALIFIED(__) then
      dumpPathLastIndent(path)
    case IDENT(__) then
      '<%name%>'
    else
      errorMsg("dumpPathLastIndent: Unknown path.")
    end dumpPathLastIndent;

template errorMsg(String errMessage)
  ::= let() = Tpl.addTemplateError(errMessage)
   <<
     <%errMessage%>
   >>
edumpErrMsg;
end DAEDumpTpl;
C User Guide

1. Install OpenModelica Compiler. Follow the instructions found in
https://www.openmodelica.org/index.php/developer/source-code

2. Compiling a model and generate XML code

The following step compiles a model to an XML code.

- Open the mingw terminal if you are a windows user and normal terminal for linux user
- In the terminal window go to the path where your model file found(C:/<%path to .mo file%>).
- Go to omc path (<%path to omc%/omc) and write the flag +s
  +simCodeTarget=XML <%your.mo file%>.mo

This is demonstrated in the following example, where +s +simCodeTarget=XML is the flag specific for XML code generation and Array1.mo is the model name.

Once compilation has completed successfully an XML file will have been generated and can be found in the same directory as your model found.
References

På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Alachew Shitahun