If we only had used XML...

Ulf Reisenbichler Hansjörg Kapeller Anton Haumer
Christian Kral Franz Pirker Gert Pascoli
Arsenal Research
Giefinggasse 2, 1210 Vienna, Austria

phone +43–50550–6606, fax +43–50550–6595, e-mail: hansjoerg.kapeller@arsenal.ac.at

Abstract

The first part of this paper will provide a general overview over XML technology and will discuss some aspects of designing applications of XML in respect of storing and retrieving data in a standardized and efficient way. This overview will focus on the essential elements of XML needed for data modeling. The second part will introduce a dynamic link library based implementation for XML data handling out of Dymola.

Consequently these considerations should lead into a discussion of how to set up standardized applications of XML for modeling and handling data in various domains of engineering to benefit from the capabilities of XML technology for data exchange in general and specifically for data handling in simulation environments like Dymola.

Keywords: XML; data modeling; data handling; data interchange; XML schema; XSD

1 Introduction

Is there an answer to the recurring question of how to store data for easy and efficient data handling and data interchange? Do we have to continue to pick and poke bits and bytes out of unreadable data chunks, or is it possible to retrieve data much more convenient? Is there a way to design a language to describe data and data structures for our own specialized purposes? Will anybody intuitively understand what we tried to formalize? The response to all of these questions would be: Yes ... if we only had used XML.

In addition to structure and interpretation of data, storing and retrieving data either in plain text or binary formats will always cause explanatory effort of *how* these data are stored – e.g. position of characters or bytes. This fact necessitates the development of individual parsing algorithms for each data format. Using the standardized structure of XML will place emphasis on the data itself and their coherency. XML documents can be read and intuitively understood by humans without any additional information as well as processed by computers using already existing analysis tools.

The objective of this paper is to motivate – once more – the general use of XML for data handling [5]. The development of ModelicaXML [3] – a representation of Modelica [1] source code using XML – points out that XML is the right concept. A brief theoretic outline will clarify terms and concepts without getting to deep into theoretical computer science or formal syntax and semantics. An interdisciplinary example will demonstrate how to interface XML data sources out of Dymola, and will show: Using XML is easy, intuitive and the best investment for the future.

2 XML

2.1 History

XML (Extensible Markup Language) was derived from the 1986 standardized SGML (Standard Generalized Markup Language) going back to the late 60ies GML (Generalized Markup Language). The XML 1.0 Recommendation was published first on 10th February 1998 by the World Wide Web Consortium (W3C). The most current version XML 1.1 was published on 4th February 2004 [7], [8].

These recommendations were – more or less accurate – realized within different applications of XML. Thereby XML evolved over the past few years into a widespread standard for handling data and documents. Within this evolution various techniques and standards for processing XML documents were developed and implemented which can be easily adapted for specific needs.

2.2 General

The main idea of all markup languages is to separate the content of a document from its structure. Whereas the "structure" can represent the formatting of a document – as in (X)HTML ((Extensible) HyperText Markup Language) – or the informational content of data within a document – as in e.g. MathML (Mathematical Markup Language), an XML based encoding standard describing mathematical notation. The inherent difference between these applications of the concepts and the generalized standards of markup languages is that these standards are metalanguages. A metalanguage like XML provides the opportunity to formulate "new languages" following specific newly defined rules in accordance to the restrictions given by the standardized stipulations.
Defining a "new language" is strictly speaking the only task when creating an application of XML. This language can be very simple or highly complex. All languages must have certain "meaningful" elements – the tokens – and rules which describe the valid sequences and hence restrict the possible combinations of these elements – the syntactic rules or syntax. Mapping the patterns generated by these rules to a structural model and projecting the elements into this derived representation of the model will give a "sentence" of this language.
Within XML the generated "sentence" would be the resulting "file" derived from the freely defined syntax and tokens of the defined markup language, i.e. the application of XML, representing the ruleset for generating and understanding documents following these rules. But XML is not simply the resulting "file", it is the sum of the concepts and utilization of the surrounding technologies.

2.3 Components

This section will provide a synopsis of the basic concepts and some components within XML technology to demonstrate how they interact one with each other unfolding the power of XML.

2.3.1 Elements

The token elements of XML are represented by the characters enclosed in angle brackets, which mark the characters surrounded by the combination of a start tag `<x>` and the corresponding end tag `</x>` – with '`/`' as symbol for ending the tag – as their content, whereas `x` is the name of the token. The content of an element may be solely character data or other elements or both of them.
A token without content can be written as `<x/>`, the empty element tag. Tokens can be specified further with attributes of the form `attribute="value"`. Attributes may only appear in start tags and empty element tags, e.g. `<x attribute="value">`, and must be unique.

2.3.2 Documents

An XML document is the data structure which is normally denoted as XML file when saved to a persistent storage device. Combining the tokens to a document only two structural conditions must be kept:

1. Every document must have a root element
2. All tags must be properly nested

These conditions will build the structural model of a tree. This is often stated as being the syntax of XML. But using a tree as structural model is not applying syntactic rules to that model. The model only gives the structure on which the syntax will operate. If a document meets this condition, it is well-formed and therefore an XML document. By definition a *not* well-formed document is *no* XML document.
Beside this tree of tokens an XML document can include other non token elements. Usually an XML document starts with an XML declaration of the form `<?xml version="1.0" encoding="UTF-8"?>`. The attribute `version="1.0"` is mandatory. The optional attribute `encoding="UTF-8"` declares the used character set and should always be used to ensure correct cross locale processing. A comment can be added between `<!--` and `-->` outside the tags anywhere in the document. There are some more elements defined by the W3C recommendation but will not be discussed here.

2.3.3 Parsers

The core piece of machine processing XML documents is an XML parser implementing all the features needed for a specific task. None of the currently available parsers – such as Xerces, libxml, Saxon or Microsoft MSXML – is implementing all of the features recommended by the W3C up to 100% [2]. While parsing mechanisms can be very different, the interface presenting the content of an XML document to an application will follow one of the standard parsing

models, i.e. the Simple API for XML (SAX) [4] or Document Object Model (DOM) [6].
The Simple API for XML is an event driven parsing model. The underlying parsing algorithm informs an application of certain events representing the structure of an XML document through callbacks. These callbacks will indicate the start and end of the document, the start and end of an element, the content of the elements and others. As SAX simply takes the document as a stream to read through without saving any content in memory it is most commonly used in performance critical applications or when parsing large documents.
The Document Object Model is the model of choice for analysis and alteration of XML documents. The parser will create a complete representation of the document as a DOM tree in memory. This tree can be traversed to retrieve information or may be transformed in structure or content. DOM is, in contrast to SAX, processing time intensive and memory consuming when parsing the document, but once loaded into memory – which is not possible with SAX as such – all operations will not be very time consuming and the results of these operations can be saved to a persistent storage device.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<motor serial="M-453123">
 <type>asynchronous induction machine</type>
 <stator part="s-4583/733">
  <material>IronSheets</material>
  <slots>
   <quantity>48</quantity>
   <slot>
    <dimension>
     <height>23e-3</height>
     <width>7e-3</width>
    </dimension>
    <winding>
     <material>Copper</material>
    </winding>
   </slot>
  </slots>
 </stator>
 <rotor part="r-4877/A72">
  <material>IronSheets</material>
  <slots>
   <quantity>40</quantity>
   <slot>
    <dimension>
     <height>20.6e-3</height>
     <width>2.8e-3</width>
    </dimension>
    <winding>
     <material>Aluminium</material>
    </winding>
   </slot>
  </slots>
 </rotor>
</motor>
```

Figure 1: motor.xml

2.3.4 Evaluation

XML provides two levels to evaluate the structural correctness of documents. A document is well-formed if the structural standard requirements of XML are met, i.e. the document simply must be parseable. To meet the stronger constraint of being "valid" the document must pass a check against additional restriction rules. These additional rules can be defined either as Document Type Definitions (DTD) or as XML Schema Definitions (XSD) each representing constraints upon what elements may appear in a document, their relationships to each other or what types of data are represented by them [12]–[16]. DTD will not be discussed here as XSD is the much more powerful and most recent standardized schema definition.
Figure 1 shows a simple well-formed XML document, with token elements projected onto an XML tree structure. Some of these tokens contain other tokens and some contain values. Intuitively humans will construct a hypothesis of the structural rules, i.e. the syntax, and the "meaning", i.e. the semantics, of these tokens.
The semantic content of these elements is the assigned value determined by the token. In this example values are only assigned to the tokens which are leaves in relation to the tree structure, i.e. which terminate their branch of the tree. On the other hand the interaction of terminal and non terminal elements is bearing intrinsic semantic information, i.e. this "tokenized" structure reflects the meaning of the syntactically defined branches. To give a precise definition of this restrictions on values and possible syntactic sequences of the tokens – either for humans or machine processing – and to further check this definitions against an XML document an XSD document can be set up (fig. 2), which then will verify the content of this XML document by the validating mechanism of the parser.

The XML Schema definition language opens the possibility to declare XML elements in terms of type definitions. These element declarations, when applied to an actual XML document, represent the abstract concept of type and token. The type of a token will restrict sequences of tokens and define their content. The limited number of components in this example will only sketch the potential of the XML Schema definition language, but illustrates the principles of its conception. The XML `element` with the `name` `motor` is a `complexType` which may only contain a `sequence` of the elements `type`, `stator` and

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:material="http://www.arsenal.ac.at/material"
       xmlns:format="http://www.arsenal.ac.at/formats">
  <xsd:import namespace="http://www.arsenal.ac.at/material"
             schemaLocation="material.xsd"/>
  <xsd:import namespace="http://www.arsenal.ac.at/formats"
             schemaLocation="formats.xsd"/>
  <xsd:element name="motor">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="type"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="stator" type="activepart"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="rotor" type="activepart"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="serial" type="format:serial"
                     use="required"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="activepart">
    <xsd:sequence>
      <xsd:element name="material" type="material:magnetic"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="slots" type="slots"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="part" type="format:part"
                   use="required"/>
  </xsd:complexType>
  <xsd:complexType name="slots">
    <xsd:sequence>
      <xsd:element name="quantity" type="xsd:integer"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="slot" type="slot"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="slot">
    <xsd:sequence>
      <xsd:element name="dimension" type="format:size"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="winding" type="winding"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="winding">
    <xsd:sequence>
      <xsd:element name="material" type="material:conductor"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Figure 2: motor.xsd

`rotor`, and has an attribute named `serial`. Within the sequence the occurrence of the elements is further restricted by the numbers given in the attributes `minOccurs` and `maxOccurs`, i.e. `type` may, while `stator` and `rotor` must occur exactly one time. These are purely syntactic rules. The attribute `type` now gives the additional information about the content of the elements. The elements `stator` an `rotor` are of the same type `activepart`, which represents on his part a sequence of `material` and `slots`, and the required attribute `part`. This new subordinated structural sequence is also defined purely syntactically, but the succession of elements assigns a meaning to the whole branch, which could be denoted as syntactico-semantic rule. The element named *type* has no attribute `type` and therefore can have any content being a *value* as it is a terminal element lacking any further branching.

Additionally, fig. 2 illustrates the concepts of XML namespaces and qualified names. An XML namespace is defined with the attribute `xmlns` relating an URI reference, e.g. `http://www.w3.org/2001/XMLSchema`, to XML elements, which then build up a collection of names belonging to this namespace. A namespace may be defined in any XML element – not only in XSD schemas – incorporating all subordinate elements into this namespace. To delimit the scope of the specified namespace a prefix, `xmlns:prefix`, may be assigned to qualify only certain elements as belonging to this namespace. The resulting qualified name of the element, `prefix:name`, binds the local name of the element to the specific namespace thus the same name can be used within different namespaces.

An XML Schema may also be compound of different schemas which can be defined as `import` into a schema. The imported schemas must each define their own `namespace` and may have a given `schemaLocation`. These namespaces and their prefixes are defined in the root element `schema`, here: `xsd:` the namespace of the XML Schema definition itself, `material:` the namespace of a schema defining different "materials" (fig. 3) and `format:` the namespace of a schema defining different number formats (fig. 4).

In fig. 2 the two syntactically defined elements named `material` – one child to `activepart` and the other child to `winding` – are represented by different types, i.e. `material:magnetic` and `material:conductor`, specified in fig. 3. The `xsd:simpleType` named `magnetic` of the base type `xsd:string` underlies the `xsd:restriction` that its content can only be one value out of the `xsd:enumeration` containing `IronSheets` and `CastIron` – correspondingly `material:conductor`. A simple type is always a terminal element, but its value and therefore its semantic content may be restricted.

The values of an element or an attribute may also be restricted using a given `pattern` like a regular expression or assigning a simple type like one of the built in data types of the XML Schema definition language, e.g. `xsd:integer`, `xsd:string` or `xsd:float`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.arsenal.ac.at/material">
  <xsd:simpleType name="magnetic">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="IronSheets"/>
      <xsd:enumeration value="CastIron"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="conductor">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Copper"/>
      <xsd:enumeration value="Aluminium"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 3: material.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.arsenal.ac.at/formats">
  <xsd:complexType name="size">
    <xsd:sequence>
      <xsd:element name="height" type="xsd:float"
                   minOccurs="1" maxOccurs="1"/>
      <xsd:element name="width" type="xsd:float"
                   minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="serial">
   <xsd:restriction base="xsd:string">
     <xsd:pattern value="[M][\-][0-9]{6}"/>
   </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="part">
   <xsd:restriction base="xsd:string">
    <xsd:pattern value="[r|s][\-][0-9]{4}[/][A-Z,0-9]{3}"/>
   </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Figure 4: formats.xsd

2.3.5 Retrieving data

Besides others, XPath (XML Path Language) is a standard query language for data retrieval within an XML document [11]. XPath expressions represent a specific node or node set within an XML document. Evaluating an XPath expression against a document, a parser will return all matching nodes for the structural description given. The expression `/motor/rotor/slots/quantity/text()` which defines the complete path to the second `<quantity>` node in fig. 1 will return the value, i.e. 40, of the node using the XPath function `text()`. The expression `//slots/quantity/text()` – with “`//`” meaning get all nodes matching within the document no matter where they are – will return 48 and 40. The complete XPath language gives much more possibilities for creating query expressions, but these very basic expressions illustrate the intuitive understanding and the simplicity of XPath.

2.3.6 Displaying data

To present data in an actually human readable format, a parser may transform an XML document into a HTML document using the Extensible Stylesheet Language (XSL) [9], [10], [17]. XSL uses XPath to retrieve data out of the XML document, which will be displayed after transforming a `template` into the resulting HTML representation. Figure 5 shows a minimal `stylesheet` displaying the `xsl:value-of` the Xpath expression `/motor/stator/material` in a minimal HTML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/">
  <html><body>
   <xsl:value-of select="/motor/stator/material"/>
  </body></html>
 </xsl:template>
</xsl:stylesheet>
```

Figure 5: motor.xsl

All kinds of documents for validating (XSD) and formatting (XSL) can be bound to the XML document within itself (fig. 6). These files may be distributed over the www. If the parser (within a browser) is interpreting all documents and transformations correctly the XML document will be self-testing, self-validating and self-formatting.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="motor.xsl"?>
<motor serial="M-453123" xmlns:xsi="http://
              www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="motor.xsd">
```

Figure 6: Connected documents

2.4 XML-Document Design Guidelines

Designing a complete markup language for a specific application area will be, in any case, a laborious task. The components introduced in the previous sections imply the elementary principles for designing the base units of such an application, i.e. the typed elements represented by tokens when building up a document. Types can either be represented by a terminal element or another structural element. Grouping the structural elements will end up in the complete document.

```
<p>
  text in a
  <table border="1">
    <tr>
      <td bgcolor="red">terminal</td>
    </tr>
  </table>
  non terminal element
</p>
```

Figure 7: Content centric XML

The design of the elements is intimately connected with the specific application of XML. If the main focus of a metalanguage is put on adding information to the "pure" content of a document, as in (X)HTML, this will be a content centric approach.
If the structure and the values of the data themselves will be focused, as if XML is used for data handling, this will be a data centric approach. The XHTML code snippet in fig. 7 illustrates what should be avoided when using XML for data handling. The paragraph element `<p>` contains besides its structural sub element `<table>` other character data which are "values" in a stricter sense. Comparably the attributes `border` and `bgcolor` contain values not specifying their structural information but assigning additional content to their "real value".
When following the data centric approach a clear distinction of what is or can have a value and how the data are structured must be made when formalizing and hence abstracting the real world. Thus the structural elements, i.e. non terminal tokens, should strictly reflect the conceptual context of the data. Values should exclusively be represented by the content of terminal elements and never within attributes. Attributes may be used to identify elements within the structural relation or specify external references not directly related to concepts within a specific document. This will transform fig. 7 to – for example – fig. 8.

This will lead to a strict structural definition of the values within a document, and will give clearly defined and simple XPath expressions when retrieving data.

```
<p>
  <text>terminal</text>
  <table>
    <format>
      <border>1</border>
    </format>
    <content>
      <tr>
        <td>
          <color>
            <background>red</background>
          <color>
          <text>terminal</text>
        </td>
      </tr>
    </content>
  </table>
  <text>terminal</text>
</p>
```

Figure 8: Data centric XML

3 Dymola XML

3.1 XML Engine

The architecture of the XMLEngine environment (fig. 9) is designed as a scalable "open" system of components which can be used out of multiple applications. The central unit within this environment is the dynamic link library XMLEngine.dll. It is implemented in C++ and can be used directly by applications dynamically loading the library and calling the interface methods.

The XMLEngine library has no internal parser and therefore must be bound to an external parser. The current implementation uses Xerces 2.6 and its XPath module Xalan 1.9 interfaced by the dynamic link library Xerces.dll binding the XML functionality to the application.

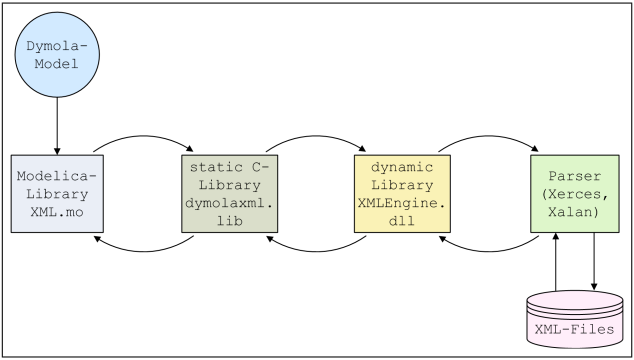

Figure 9: Architecture of the XMLEngine environment

3.2 Dymola

Following the object oriented approach of Modelica, the programming language incorporated into the simulation environment Dymola, the idea of strict separation of code (models) and data (parameters) reflects this attempt for data handling. Using XML documents opens the possibility to store and retrieve primitive data types as well as highly complex data objects in a flexible way.
If a defined application of XML is used as base for data exchange, the import of precalculated data, for e.g. a thermodynamic model, and the export of results, e.g. state variables, for any calculation and simulation tool, following these standardized rules, is guaranteed to work correctly with minimum effort.
Dymola is bound via the static link library dymolaxml.lib for MS-Visual C or libdymolaxml.a for GNU C to the XMLEngine environment. The XML query language XPath is used to import data from XML documents and to export data into predefined template documents. The integration of the XMLEngine environment into Dymola in cooperation with the scalability and flexibility of XML opens a wide range of possibilities for data handling and data exchange.

3.3 Interdisciplinary Application Example

An interdisciplinary example (containing mechanics, electrical engineering and thermodynamics, fig. 10) taken from drive engineering will demonstrate how to retrieve parameter values from an XML document and how to save results to a file using a predefined XML document template. The single aspects of this example and the corresponding parameter values are reflected by the structure of the elements in the underlying XML document (fig. 11). The different elements are combined via connectors forming the entire model (fig. 10).

- “thermalModel” represents the thermodynamic activity in a permanent magnet DC motor
- the permanent magnet DC motor is as such an electromechanical model
- the electric source represents an electrotechnical application
- the mechanical load is characterized by pure mechanical equations

Having defined the data in an XML document – and not within each single sub model – different aspects

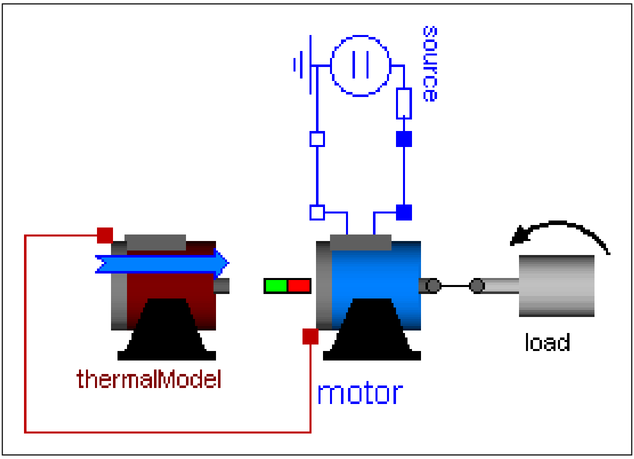

Figure 10: Interdisciplinary application example

of the entire model may be simulated loading the data from – and manipulating the data in – an external data source. By changing the parameter sets – in the now “centralized” data source – it will be possible to analyze for example the thermal stress within the field windings of different motor types (fig. 15). More elegantly different parameter sets may defined directly in the XML document and selected within Dymola. This approach is illustrated here by the “load_data” element (fig. 11) to operate several workloads of the motor.
The Dymola XML Library is using the Modelica standard external function interface to load and access data from an XML document. For loading data into a model firstly the parser must be initialized to obtain a handle for further processing (fig. 12): `domParserHandle := createDOMParser( ParserName )`
The next step is loading the XML document as DOM tree representation into memory, using the handle of the previously created parser: `DocumentHandle := loadDocument( domParserHandle, FileName )`
When initializing the parser and loading documents within the same model this must be done in an initial algorithm to ensure strict algorithmic processing order.
Now having a handle to a document retrieving and changing of the data is possible using XPath expressions. In this model the parameters and the initial conditions for the complete characterization of the system of differential equations will be retrieved from the DOM trees residing in memory. The XML document template for storing the results of the simulation will be loaded for later use. To obtain a single result value when evaluating an XPath expression against the

```
<?xml version="1.0" encoding="US-ASCII"?>
<data>
  <source_data>
        .
        .
        .
  </source_data>
  <motor_data>
    <motor name="Standard">
      <electric>
        <VaNominal>  100</VaNominal>
        <IaNominal>  100</IaNominal>
        <rpmNominal>1425</rpmNominal>
        <Ra>  0.05</Ra>
        <Ta>    70</Ta>
        <La>0.0015</La>
        <J>   0.15</J>
      </electric>
      <thermal>
        .
        .
        .
      </thermal>
    </motor>
  </motor_data>
  <load_data>
     <load name="NoLoad">
        .
    </load>
    <load name="Constant">
        .
    </load>
    <load name="Intermittent">
        .
    </load>
  </load_data>
</data>
```

Figure 11: IDE.xml

```
Modelica definition
model Example
  parameter String sourceName = "Standard";
  parameter String motorName = "Standard";
  parameter String loadName = "Intermittent";
.
.
.
  parameter TAmb = 20 "Ambient temperature";
  output TWinding=thermalModel.Winding.T;
  output THousing=thermalModel.Housing.T;
 protected
  parameter Integer domParserHandle(fixed=false);
  parameter Integer parDocumentHandle(fixed=false);
  parameter Integer iniDocumentHandle(fixed=false);
  parameter Integer trmDocumentHandle(fixed=false);
.
.
.
initial algorithm
  domParserHandle :=createDOMParser("Xerces");
  parDocumentHandle :=loadDocument(domParserHandle,
                        "IDE.xml");
  iniDocumentHandle :=loadDocument(domParserHandle,
                        "Initial.xml");
  trmDocumentHandle :=loadDocument(domParserHandle,
                        "TerminalTemplate.xml");
  if iniDocumentHandle == 0 then
    T0Winding:=TAmb;
    T0Housing:=TAmb;
  else
    T0Winding:=getReal(iniDocumentHandle, "//data/
                        TWinding");
    T0Housing:=getReal(iniDocumentHandle, "//data/
                        THousing");
  end if;
.
.
.
end Example;
```

Figure 12: Initial algorithm

loaded document the handle of the document and the XPath expression has to be given over to the corresponding function:

`getReal( DocumentHandle, XPath )`

This function returns the first result matching the XpathExpression (fig. 13).

All functions are also available for Integer, Boolean and String values. All "get" functions are also available in versions evaluating the second XPath expression starting at the result from a context node's XPath result: `getRealFromContext( DocumentHandle, ContextNodeXPath, XPath )`

Changing data values in previously loaded documents one of the "set" function will be called specifying the document, the location as XPath expression and the new value:

`setReal( DocumentHandle, XPath, newValue );`

Finally the changed template document will be saved to disk calling (fig. 14):

`saveDocument( DocumentHandle, newDocumentName );`

The following simulation results refer to the above mentioned DC machine operated at continuous duty with intermittent periodic loading. The load torque is alternating between no load (0 Nm, 24 s) and load (80 Nm, 36 s) periodically. In fig. 15 the temperature rise of the DC-machine field winding and in the housing is depicted. In a new simulation procedure these final values can be used as new initial values.

4 Conclusions

Separating simulation models and data can be very convenient when using XML documents and XPath expression for storing, retrieving and manipulating

```
Modelica definition
model DC_PM "Permanent magnet DC machine"
  parameter Integer parDocumentHandle = 0;
  parameter String motorName = "";
  extends
.
.
.
protected
  parameter Modelica.SIunits.Voltage VaNominal =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    VaNominal/text()");
  parameter Modelica.SIunits.Current IaNominal =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    IaNominal/text()");
  parameter Modelica.SIunits.Conversions.
    NonSIunits.AngularVelocity_rpm rpmNominal =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    rpmNominal/text()");
  parameter Modelica.SIunits.Resistance Ra =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    Ra/text()");
  parameter Modelica.SIunits.Conversions.
    NonSIunits.Temperature_degC Ta =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    Ta/text()");
  parameter Modelica.SIunits.Inductance La =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    La/text()");
  parameter Modelica.SIunits.Inertia J =
    getReal(
    parDocumentHandle, "//data/motor_data/
    motor[@name=\""+motorName+"\"]/electric/
    J/text()");
public
.
.
.
equation
.
.
.
end DC_PM;
```

Figure 13: Motor specification

```
Modelica definition
model Example
.
.
.
initial algorithm
.
.
.
equation
 when terminal() then
  setReal(trmDocumentHandle,"//data/TWinding",
          TWinding);
  setReal(trmDocumentHandle,"//data/THousing",
          THousing);
  saveDocument(trmDocumentHandle,"Terminal.xml");
 end when;
.
.
.
end Example;
```

Figure 14: Storing simulation data

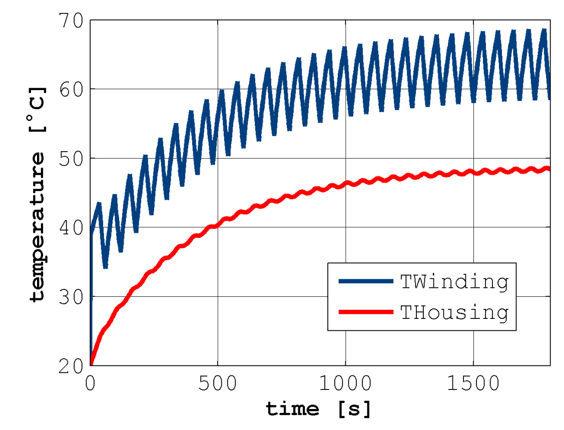

Figure 15: Simulation results

data. Additionally, XML technology provides the opportunity of modeling data in a standardized way. The resulting application of XML, i.e. the specific meta-language, can be designed independently from the use in certain models and therefore provide a consistent basis of terminology and structured data representations for various domains of engineering.

References

[1] Fritzson P., *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Piscataway*, NJ: IEEE Press, 2004,

[2] Harold, E. R., *SAX Conformance Testing*, XML Europe 2004 Conference,

http://www.idealliance.org/papers/dx_xmle04/papers/03-06-02/03-06-02.pdf

[3] Pop A., Fritzson P., *ModelicaXML: A Modelica XML Representation with Applications*, Proceedings of the 3rd International Modelica Conference, pp. 419-430, 2003

[4] The SAX project, home page, http://www.saxproject.org

[5] Tiller M., *Implementation of a Generic Data Retrieval API for Modelica*, Proceedings of the 4th International Modelica Conference, pp. 593–602, 2005

[6] World Wide Web Consortium (W3C), *Document Object Model (DOM)*, http://www.w3.org/DOM

[7] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.0* (Third Edition), W3C Recommendation 04 February 2004, http://www.w3.org/TR/2004/REC-xml-20040204

[8] World Wide Web Consortium (W3C), *Extensible Markup Language (XML) 1.1*, W3C Recommendation 04 February 2004, http://www.w3.org/TR/2004/REC-xml11-20040204

[9] World Wide Web Consortium (W3C), *Extensible Stylesheet Language (XSL)*, Version 1.0, W3C Recommendation 15 October 2001, http://www.w3.org/TR/xsl

[10] World Wide Web Consortium (W3C), *The Extensible Stylesheet Language Family (XSL)*, http://www.w3.org/Style/XSL

[11] World Wide Web Consortium (W3C), *XML Path Language (XPath)*, Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[12] World Wide Web Consortium (W3C), *XML Schema Part 0: Primer* Second Edition, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-0-20041028

[13] World Wide Web Consortium (W3C), *XML Schema Part 1: Structures* Second Edition, W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028

[14] World Wide Web Consortium (W3C), *XML Schema Part 2: Datatypes* Second Edition,W3C Recommendation 28 October 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028

[15] World Wide Web Consortium (W3C), *XML Schema 1.1 Part 1: Structures*,W3C Working Draft 30 March 2006, http://www.w3.org/TR/2006/WD-xmlschema11-1-20060330

[16] World Wide Web Consortium (W3C), *XML Schema 1.1 Part 2: Datatypes*,W3C Working Draft 17 February 2006, http://www.w3.org/TR/2006/WD-xmlschema11-2-20060217

[17] World Wide Web Consortium (W3C), *XSL Transformations (XSLT)*, Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/1999/REC-xslt-19991116