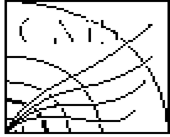
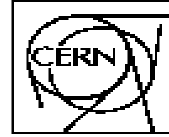
CMS NOTE 2003/031



The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



Migration of the XML Detector Description Data and Schema to a Relational Database

Asif Jan Muhammad[1,3]
Martin Liendl[1,] , Frank van Lingen[2], Arshad Ali[3] ,
Ian Willers[1]

Abstract

This document discusses the design and implementation of software for migration of the XML detector description data to a relational database maintaining the semantic structure. Relational databases offer great flexibility for data consistency, manipulation and retrieval and offer a very well developed and easy to use query language. XML on the other hand offers great flexibility for data description and is a platform independent data format but lacks very efficient data selection and retrieval mechanisms. Relational databases also allows for management on the data level, while XML limits data management to the file level. The migration software tries to achieve migration between XML and the relational format in an effort to achieve the “best of both worlds”.

[1] European Laboratory for Particle Physics (CERN), Switzerland
[2] California Institute of Technology (CALTECH), United States of America
[3] National University of Science and Technology, Pakistan

1 Introduction

In recent years XML[1] has become a de facto standard for data exchange; its nested structure provides a very flexible way to describe data. Relational database systems on the other hand offer a variety of services to developers and users including security, transaction management, and data management. They also have an optimised language for data selection and retrieval [2]. XML due to its text based format lacks efficient mechanisms for data selection and retrieval. A combination of XML and relational formats offer a promising strategy for effective data description, exchange and management. It is therefore useful to study and implement mechanisms that are able to interchange data between XML and relational formats; this will help capitalizing on the strengths of both data formats and provide a useful means to integrate already existing data.

Data migration can be migration of data to a different persistency format while maintaining the semantic structure of the data or a complete transformation of the semantics and persistency format. In the first form the schema remains unchanged and the migration software tries to exploit the advantage of another persistency format in terms of scalability, data retrieval efficiency or stability , where as in the second form migration tries to ensure effective mapping between distinct information representation domains.

"XMLized" data have gained acceptance within CERN. Currently XML is being used as a mechanism to describe the detector in a consistent and coherent manner and to serve as a common source of information for CMS offline software [3] [4][5]. Both XML files and relational databases offer advantages for the management and manipulation of the Detector Description data [4]. XML files can serve as a mechanism for physicists to insert or edit detector description data and serve as data exchange format [6]. Relational databases offer efficient data storage, retrieval and management.

This note discusses the migration from the XML detector description data to a relational model maintaining the semantic structure of the data. It involves mapping the XML Schema to a corresponding relational schema and reading data from XML instance documents to populate the relational database. The migration does not assume any specific schema format and is independent of the DDL Schema [4] .It can also handle schema evolution by either introducing new tables i.e. if new "types" are introduced or by changing the already existing tables if the structure of corresponding "types" is altered.

The rest of the note is organized as follows. Section 2 discusses the requirements for performing the mapping between the XML model and the relational model. Section 3 contains a comparison between the relational model and the XML Schema model. Section 4 describes the architecture. The design is discussed in section 5 Section 6 discusses the implementation, the DDL SQL data model, and MySql client. A comparison of size, query turn around time and syntax between XML and Relational data model is highlighted in section 7. Section 8 contains an overview of the related work, and section 9 concludes the paper and present some ideas about future work.

2 Requirements

The current implementation of the detector description software (DD-SW) for the offline applications, simulation and reconstruction parses XML files containing description data. Description data is converted to a transient object model within the software [5]. An API is exposed to and used by the client software providing various services commonly needed. The transient model and the provided services are independent of any data storage mechanism. Furthermore, DD-SW provides an API for building its transient model, which is used by the XML parser for binding. Besides the capabilities of the parser , which are loading XML from a file system and retrieving XML via http, no other means of data retrieval from storage is currently available.

The XML format for detector description [4] can be used as a description format (and used to store data in ASCII files) and as an exchange format. The transient detector description model is independent of the

persistency mechanism that stores the detector description data. Detector description data can be stored in, XML documents, dedicated XML databases and relational databases, these conditions lead to following main requirements:

[R1]: A persistency mechanism has to be foreseen for detector description. XML documents are the primary information source for the detector description software (DD-SW). If XML is not the primary information source for DD-SW, the transient model has to be extended with an interface to another persistent data source

[R2]: A persistency mechanism has to be foreseen for the transient model of DD-SW. It is not always required to build a full transient model, e.g. sometimes only the description (or parts thereof) of sub-detectors need to be loaded into memory. This requires a selection of input data based on parameters provided by the client software.

[R3]: It must be possible to execute queries on the persistent store in order to retrieve a consistent subset of the detector description data.

In the case of Detector Description Database (DDD), the transient model of DD-SW is mapped to an appropriate XML schema, which is used to validate the XML input data. Thus the XML schema is used for persistency model of both - XML documents and the transient model of DD-SW. A possible solution fulfilling [R1] and [R2] is therefore to provide a persistent store for schema based XML data. Data based on a relational database schema is well suited to be subjected to optimized queries using a standardized query language i.e. SQL. One of possible way to fulfill [R3] is to have a relational database as persistent store.

3 XML Schema versus Relational Schema

The W3C XML Schema Definition Language (referred to as XML schema from now on) is an XML language for describing and constraining the content of XML documents. XML schema defines the structure of an XML document. A relational schema defines the structure and constraints on the relations (tables) in the database [1]. The aim of the DBMS is to provide an environment that is convenient and efficient in storing and retrieving information (data) from the database, XML on the other hand was developed as a format to structure and exchange data and is suitable for storing small amount of data e.g. few tens of megabytes.

Detailed treatment comparing the XML schema and the relational schema is subject to a number of ongoing research projects, (see [7][8]) for a detailed comparison of both the models. Here we will highlight a few very distinct and fundamental differences between these structures.

Structuring XML documents using a schema is not mandatory for all XML documents. When the XML documents are based on a schema, this is indicated by including a reference to a schema location inside the document. So the validation of the XML documents against some predefined schema can only take place if the XML documents are using the schema. This is fundamentally different from a relational database system where an explicit definition of the schema is stored separately inside the database system and any data inserted into the database is checked against this schema.

One XML document can be constructed by referring to multiple schema documents. This enhances the reusability of already defined structures. This is different from the relational schema where every relation (i.e. table) has a fixed schema and does not support the concept of “including” structures or elements from other relations or tables.

Applications and programs reading XML documents don’t need prior knowledge about the structure of the XML document. They can discover the structure of the documents at runtime. However, to read any data from a relational database system, the application needs to have prior knowledge of the schema.

Relational schema are modeled around the concept of relations and attributes. Attributes can have a domain i.e. set of possible values, and relations contain data defined as per the schema. XML Schema documents

contain element and attribute declarations having associated type definitions. These type definitions control the data that the elements and attributes can hold, thus serving as domains for these structures.

The relational schema can define certain columns in a table to be optional and SQL type "NULL" appears for these columns in the table. The XML schema can define certain attributes and elements are optional in the instance documents, but unlike relational schema these elements and attributes do not require a special "NULL" value as a place holder in the instance documents.

In a relational structure relationships can be expressed between various tables by means of foreign keys but foreign keys are restricted to the same database, in an XML schema we can have references which span multiple documents and namespaces [4] [9].

Both structures have mechanisms for querying the data and selecting the required data. The query language for relational database systems, SQL, is very well developed where as for XML the language for querying and selecting the data have been developed more recently [10][9].

Relational database systems have been used for a long time and have very well developed platforms which offer facilities like transaction management, security and data consistency. In recent times we have also seen frameworks which offer similar facilities for XML data [11][12].

4 Architectural Overview

Figure 1 describes the architecture of the mapping software and the following sections give a brief overview of the components.

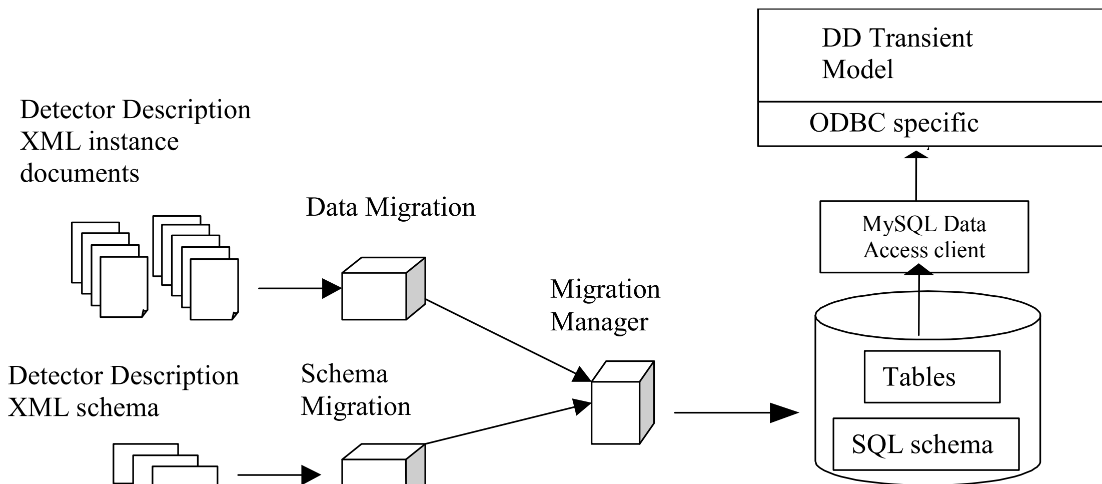


Figure 1 Architecture

4.1 Schema Migration

The Schema Migration component translates the XML schema into the corresponding relational schema. The input to the Schema Migration component is the XML schema file or a set of XML schema files. It parses the files, constructs corresponding type definitions and builds the relational schema, which is then migrated to a relational table structures. The XML Schema contains the type declarations, simple type declarations and complex type declarations[13], which are mapped onto relational table columns and relational tables respectively. The Schema Migration component not only provides mapping from types to the corresponding relational structure, it also discovers and preserves various constraints and relations expressed in the XML schema structure.

[4] Not part of W3C specifications yet

4.2 Data Migration

After performing the mapping from XML schema to relational schema, XML data based in the XML instance documents is migrated to the relational database. The data migration component reads the instance documents and migrates the data to the relational database tables.

4.3 Migration Manager

All database related tasks such as creating tables, inserting data, performing updates and selecting the data, are performed by the Migration Manager. It consists of a web component which is used to provide applications access through http and a database bean implementing the JDBC layer for data access. The database bean is useful in separating the underlying database management systems from the migrating software, thus providing flexibility to use different database management systems without changing the migrating software. The web component is useful for making the application available through web.

4.4 MySQL Data Access Client

The MySQL data access client serves as the data access interface to the DDD core software. DDD core software accesses the MySQL client to build the transient object model which is then used by offline applications and programmes to construct the detector structure .

5 Design

This section contains discussion about various design issues involved in migrating the XML schema and the XML instance documents to relational schema and relational database tables respectively.

5.1 Schema Migration

The XML schema contains several language constructs that need to be translated to a relational model. Many of the examples used in this section are taken from the detector description schema.

- **Simple Type Declarations**

Simple Type declarations are used to define content models for the attributes in the XML schema documents. Simple type declarations can be of the form:

```
<xsd:simpleType name=”densityT”>
        <xsd:restriction base=”xsd:string”/>
</xsd:simpleType>
```

Simple type declarations extend or restrict the built in types provided in the W3C Schema language, and can be used to express restrictions on the possible values or to specify patterns or range of values. All simple types of the detector description schema are restrictions of strings. Within the detector description domain the numbers need to specify the units in which the quantity is measured. For example: an attribute “angle” can have the value 40.3 degrees. Within an XML schema a built-in float type can describe: angle=”40.3”. However within the detector description domain, the units need to be specified: angle=”40.3*degrees”. So these quantities are expressed as extensions to schema built-in type “string” instead of being described as float or integer[5]. The software in the DDD core converts the strings to doubles or floats. Simple types, type float or double, can be migrated to a relational database in the same way as strings. Simple types become fields of string, float or double within a database table.

- **Complex Type Declarations**

Complex type declarations allow node elements and attributes to be declared and referred inside the type declarations, providing a mechanism to express complex content models. Element nodes can also contain arbitrary complex type definitions resulting in a nested and recursive type declaration.

---

[5] described in annex A

Within an XML Schema documents, complex and simple type declaration can be defined globally (more node elements can use the type). The advantage of global type declaration is a clear and partitioned schema document (type declaration, attribute groups and element definition). DDL has adapted this partitioned approach for its schema description. Such a partitioning of the XML schema makes it easier to migrate to a relational schema. Any XML schema can be partitioned in the type declarations, attribute group and element definition. The partitioning is a way to layout the XML schema. The current migration software needs a partitioning of the type declarations, attribute groups, and element definition to successfully create a relational schema. The migration software currently does not have the capability to resolve type declarations defined inside an element.

Complex types are mapped as database relations or tables. Complex types may also contain attributes that are in turn modeled as database table columns. Furthermore, complex types may also inherit from a base type in which case all the attributes for the base type are considered as attributes of the child type and are translated as table columns. The following paragraphs discuss the various complex types.

Complex types can contain references to attribute groups. Attributes are simple types and will generate a column in the relational database. Within an XML schema an attribute is either optional or required. Within a relational database optional/required translates to NULL/NOT NULL.

The following example shows a complex type with an attributeGroup reference and the associated SQL statement that creates the table in the relational database:

```
<xsd:element name=”Material” type=”MaterialType”/>

<xsd:complexType name=”MaterialType”>
        <xsd:attributeGroup ref=”MaterialAttributes”/>
</xsd:complexType>

<xsd:attributeGroup name=”MaterialAttributes”>
        <xsd:attribute name=”name” type=”xsd:string” use=”required”/>
        <xsd:attribute name=”symbol” type=”xsd:string” use=”optional”/>
</xsd:attributeGroup>
```

This creates the following SQL statement:

```
CREATE TABLE MaterialType(
MATERIAL_ID    integer   PRIMARY KEY AUTO_INCREMENT
name        string    NOT NULL
symbol      string    NULL )
```

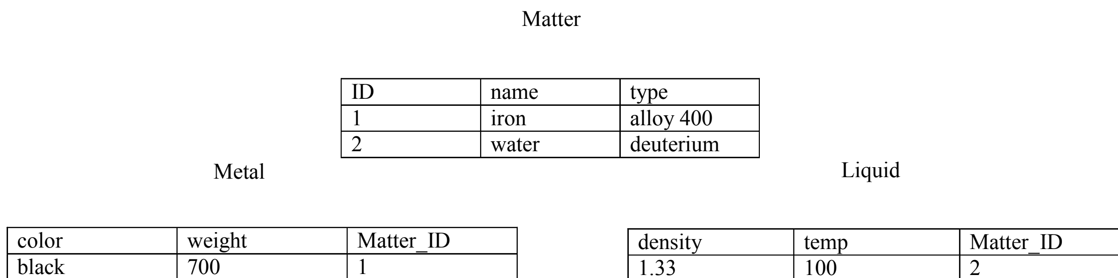
Complex types can inherit from an already defined type. The inheritance concept is similar to the inheritance concept of object orientation. Inheritance can be modeled in one of following ways. In the first approach a table for a base type can contain the common attributes for all the child types and tables, for child types model only child specific attributes, and in a second approach attributes inherited from base type are considered as attributes for the child type and modeled as columns in the table of the child type. Consider the following example;

```
<xsd:complexType name=”Matter”>
<xsd:attribute name=”name” type=”xsd:string” use=”required”/>
<xsd:attribute name=”type” type=”xsd:string” use=”optional”/>
</xsd:complexType>

<xsd:complexType name=”Metal”>
        <xsd:attribute name=”color” type=”xsd:string” use=”required”/>
        <xsd:attribute name=”weight” type=”xsd:string” use=”optional”/>
</xsd:complexType>

<xsd:complexType name=”Liquid”>
        <xsd:attribute name=”density” type=”xsd:string” use=”required”/>
        <xsd:attribute name=”temp” type=”xsd:string” use=”optional”/>
</xsd:attributeGroup>
```

First approach to handle inheritance will result in following structure (Figure 2);



Matter

ID	name	type
1	iron	alloy 400
2	water	deuterium

Metal

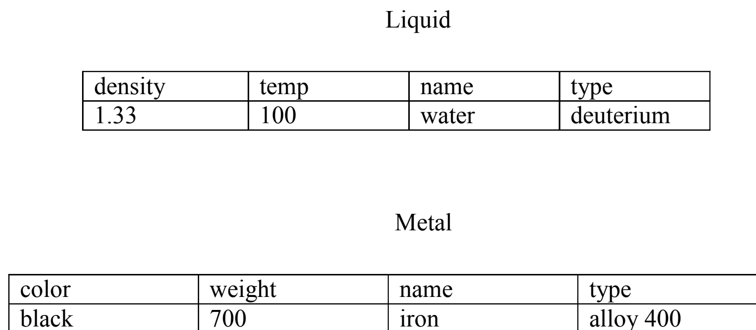
color	weight	Matter_ID
black	700	1

Liquid

density	temp	Matter_ID
1.33	100	2

Figure 2. Inheritance approach 1

In the first approach child type has an entry in the parent type table, thus resulting in dependence between child type and parent type tables. Any changes in the child type tables will have to be reflected in the parent type table and vice versa.
In order to avoid this interdependence, attributes inherited from the parent type can be considered in the same manner as attributes of the child type, this would result in the following structure (Figure 3);



Liquid

density	temp	name	type
1.33	100	water	deuterium

Metal

color	weight	name	type
black	700	iron	alloy 400

Figure 3. Inheritance approach 2

The second approach results in extra columns in the tables for child types but removes the interdependence between child and parent type tables. Every table can be updated without need for reflecting the change in other tables.

The migration software employs the second approach to model inheritance in the relational model. The attributes that are inherited from the parent element will also generate columns in the table associated with

the child type. The following example shows a complex type “ElementaryMaterialType” that inherits from “MaterialType”. Certain attribute types in the DDL are not of type string or float but simple types based on string and float.

```
<xsd:element name=”ElementaryMaterial” type=”ElementaryMaterialType”/>

      <xsd:complexType name=”ElementaryMaterialType”>
        <xsd:complexContent>
                <xsd:extension base=”MaterialType”>
                 <xsd:attributeGroup ref=”ElementaryMaterialAttributes”/>
                </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>

<xsd:attributeGroup name=”ElementaryMaterialAttributes”>
    <xsd:attribute name=”atomicNumber type=”xsd:integer”   use=”required”/>
    <xds:attribute name=”density” type=”densityT use=”required”/>
    <xsd:attribute name=”atomicWeight” type=”weight” use=”optional”/>
    <xsd:attribute name=”weightUnit type=”weightUnitT use=”optional”/>
    <xsd:attribute name=”lengthUnit type=”lengthUnitT use=”optional”/>
    <xsd:attribute name=”densityUnit” type=”densityUnitT”use=”optional”/>
</xsd:attributeGroup>
```

This leads to the following SQL statement[6]:

```
CREATE TABLE ElementaryMaterial
( ELEMENTARYMATERIAL_ID    integer   PRIMARY KEY AUTO_INCREMENT,
  atomicNumber      string    NOT NULL,
  density           string    NOT NULL,
  atomicWeight      string    NULL,
  weightUnit        string    NULL,
  lengthUnit        string    NULL,
  densityUnit       string    NULL,
  name              string    NOT NULL,
  symbol            string    NULL )
```

The previous examples showed complex types and attribute groups. It is also possible to describe complex types using references to other element nodes of certain types. These elements are represented within the relational schema as tables. The following examples show how complex types based on references to element nodes are translated to a relational SQL statement.

Node elements declared in a XML schema file have an associated type. A reference to a node element within a complex type definition implies a reference to another type definition (associated with the node element). Complex type definitions are associated to elements and are mapped to tables within a relational database. Element references within complex type definitions are mapped to references between relational tables.

There are three types of relations between entries in a relational database: one to one, many to one (one to many) and many to many. The next paragraphs discuss each of these relations.

[6] The migration software stores all numeric quantities as ‘string’ type in relational database. The DDD core software has the capability to infer correct types and values from ‘string’ types. The rational behind doing it is explained in Annex A.

- **One to one relationships**

If the migration software analyses an element definition A of type “complexType” B and finds an element reference C within the complex type definition of D of multiplicity 1, it needs to verify if the associated complex type D of this referenced element C does not contain references to elements of A with multiplicity greater than one. If there are only references of multiplicity 1 there is a one to one relationship.

The following example from the detector description XML schema shows a one to one relationship. Within the example there is a reference to an element with name “rSolid”. Within the detector description domain rSolid describes references to detector description data. Within the XML schema it is just treated as another node element. The semantics of “rSolid” (it is a reference within the detector description domain) is resolved by DDD core software.

```
<xsd:element name=”ReflectionSolid” type=”ReflectionSolidType”/>
    <xsd:complexType name=”ReflectionSolidType”>
        <xsd:complexContent>
        <xsd:extension base=”SolidType”>
                <xsd:sequence>
                    <xsd:element ref=”rSolid”/>
                </xsd:sequence>
      </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
```

The SQL statement corresponding to the above complex type definition will be:

```
CREATE TABLE ReflectionSolid(
REFLECTIONSOLID_ID    integer PRIMARY KEY NOT NULL AUTO_INCREMENT,
name varchar(50) NOT NULL          %from SolidType definition,
rSolid       varchar(50)           NOT NULL)
```

- **One to Many (Many to One)**

Within an XML schema it is possible to assign a multiplicity to an element within a complex type definition. Every element definition within the XML schema corresponds to a table definition within the relational database. Suppose there is an element definition A of type “complexType” B. Within this complex type defintion B there is a reference to an element definition C of type “complexType” D with multiplicity greater than one. Within the complex type D there is no reference to element A. Such a reference translates as a one to many relationship. The following example shows the element definition SpecPar of type SpecParType that contains a reference to a PartSelector and a Parameter (only the definition of Parameter and its type are displayed). SpecPar and Parameter consist of a one to many relationship.

```
<xsd:element name=”SpecPar” type=”SpecParType”/>

<xsd:complexType name=”SpecParType”>
    <xsd:sequence>
          <xsd:element ref=”PartSelector” minOccurs=”0”
                  maxOccurs=”unbounded”/>
          <xsd:element ref=”Parameter” minOccurs=”0”
                  maxOccurs=”unbounded”/>
    </xsd:sequence>
    <xsd:attributeGroup ref=”SpecParAttributes”/>
</xsd:complexType>

<xsd:attributeGroup name=”SpecParAttributes”>
    <xsd:attribute name=”name” type=”xsd:string” use=”required”/>
```

```
</xsd:attributeGroup>

<xsd:element name="Parameter" type="ParameterType"/>

<xsd:complexType name="ParameterType">
    <xsd:attributeGroup ref="ParameterAttributes"/>
</xsd:complexType>

<xsd:attributeGroup name="ParameterAttributes">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:attributeGroup>
```

The above XML schema snippet translates into the following SQL statements for the SpecPar and the Parameter in which the Parameter table contains a column SpecPar_ID:

```
CREATE TABLE SpecPar (
  SPECPAR_ID integer PRIMARY KEY AUTO_INCREMENT,
  name string NOT NULL )

CREATE TABLE Parameter (
  PARAMETER_ID integer PRIMARY KEY AUTO_INCREMENT,
  name string NOT NULL,
  value string NOT NULL,
  SpecPar_ID string NULL)
```

It is possible that the Parameter element is referenced within the complex type definition of SpecParType that is associated to element SpecPar but that there is also another element Par with the same complexType definition as SpecPar. This implies the creation of two tables: one table for storing SpecPar entries and another for storing Par entries. Entries within the Parameter table should be able to reference to entries in either the table SpecPar or the table Par. There are two possibilities to make references between the Pararmeter table entries and the SpecPar and Par table entries:

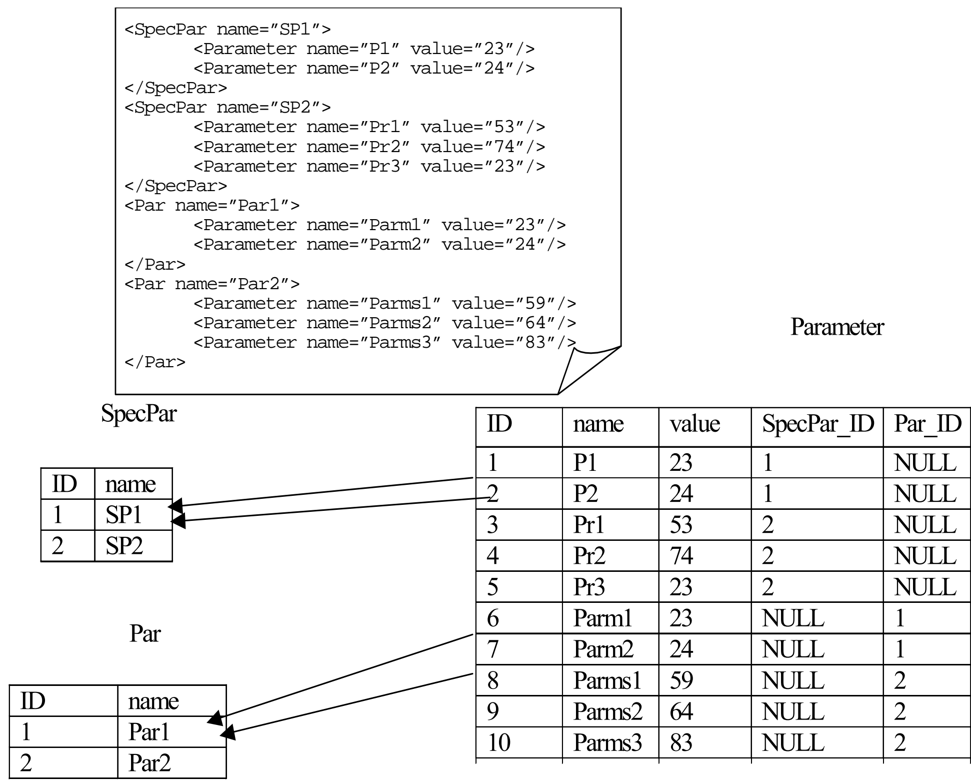
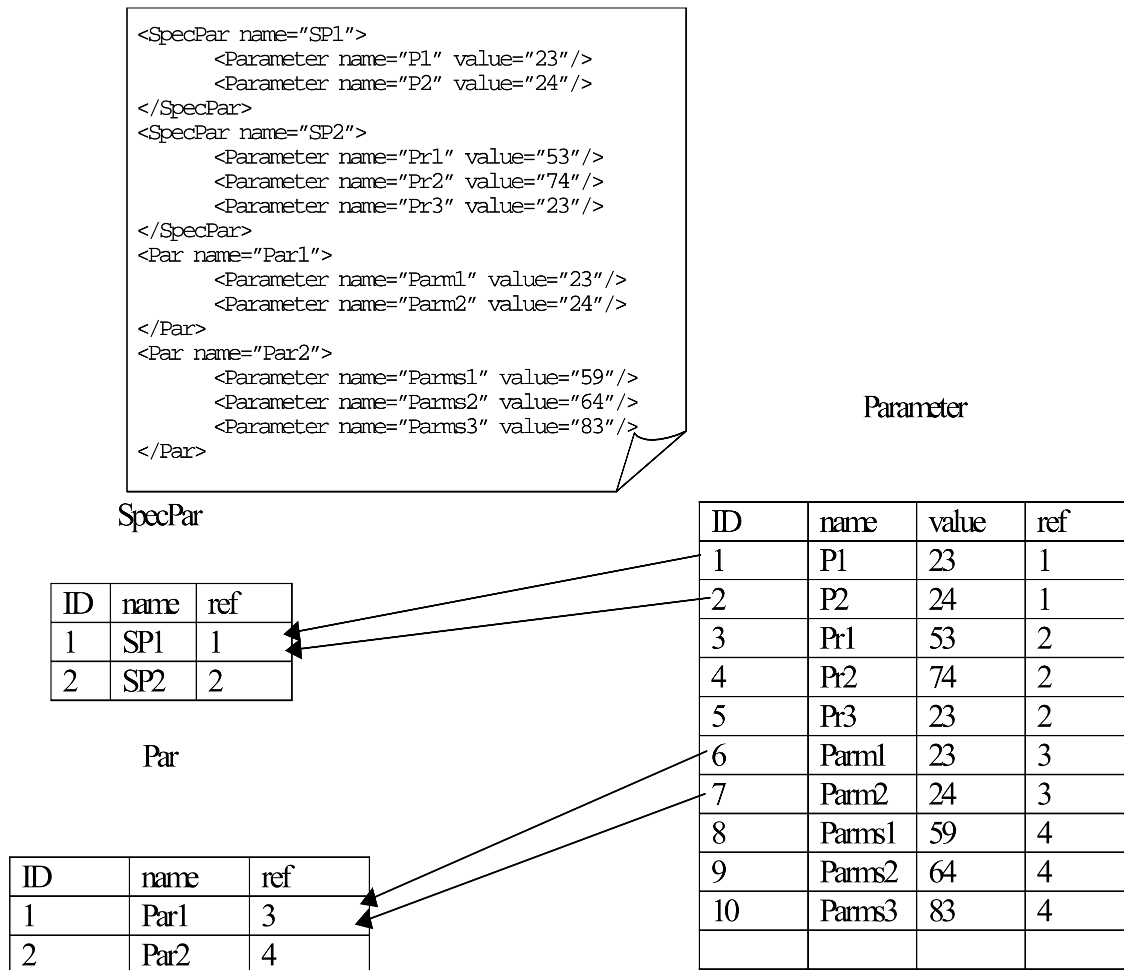


Figure 4. Two one to many examples with NULL values.

To allow references to both the SpecPar table and the Par table the Parameter Table can be extended with a column: Par_ID string NULL. When the relational database is populated with data conforming to the schema, either the SpecPar_ID will have a value or the Par_ID. The other column entry will have a NULL value. The draw back of this approach is adding an extra column to the Parameter table. Furthermore half of the entries of the combined fields of the Par_ID and SpecPar_ID columns will have a NULL value. The assumption with this approach for references is that the relational database stores only the non NULL values which means that adding an extra column does not result in extra data. Figure 4 shows an example of references with NULL values.

Another approach is not to allow Null values. Figure 5 shows an example. SpecPar and Par contain an extra column ref. The ref column contains a unique ID that is unique over the two columns. The Parameter table contains a column ref that refers to the ref Ids within the SpecPar and Par table.



```
<SpecPar name="SP1">
      <Parameter name="P1" value="23"/>
      <Parameter name="P2" value="24"/>
</SpecPar>
<SpecPar name="SP2">
      <Parameter name="Pr1" value="53"/>
      <Parameter name="Pr2" value="74"/>
      <Parameter name="Pr3" value="23"/>
</SpecPar>
<Par name="Par1">
      <Parameter name="Parm1" value="23"/>
      <Parameter name="Parm2" value="24"/>
</Par>
<Par name="Par2">
      <Parameter name="Parms1" value="59"/>
      <Parameter name="Parms2" value="64"/>
      <Parameter name="Parms3" value="83"/>
</Par>
```

SpecPar

ID	name	ref
1	SP1	1
2	SP2	2

Par

ID	name	ref
1	Par1	3
2	Par2	4

Parameter

ID	name	value	ref
1	P1	23	1
2	P2	24	1
3	Pr1	53	2
4	Pr2	74	2
5	Pr3	23	2
6	Parm1	23	3
7	Parm2	24	3
8	Parms1	59	4
9	Parms2	64	4
10	Parms3	83	4

Figure 5. Two one to many examples without NULL values.

- **Many to Many Relationships**

Within the XML schema definition of the detector description domain there were only one to one and one to many relationships but no many to many relationships. Many to Many relationships occur when a element A of complex type typeA uses a reference to element B of complex type typeB and complex typeB uses a reference to element A. The following XML schema snippet shows the many to many relationship.

```
<xsd:element name="A" type="typeA"/>

<xsd:complexType name="typeA">
    <xsd:sequence>
        <xsd:element ref="B" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="B" type="typeB"/>
<xsd:complexType name="typeB">
    <xsd:sequence>
        <xsd:element ref="A" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

A many to many relationship within XML schema would imply the creation of a table for A, B and a table describing the relationship between A and B. This would translate in the following SQL statements:

```
CREATE TABLE A(
A_ID integer PRIMARY KEY AUTO_INCREMENT)

CREATE TABLE B(
B_ID integer PRIMARY KEY AUTO_INCREMENT)

CREATE TABLE A_AND_B(
A_AND_B_ID integer PRIMARY KEY AUTO_INCREMENT,
A_ID integer,
B_ID integer)
```

- **Inline Definitions and References**

Within an XML schema it is possible to define data inside a tag or use a reference inside tag to refer to data. The reference mechanism used in DDD is very specific to the detector description format and is identified by a complex type “ReferenceType” i.e.

```
<xsd:complexType name="ReferenceType">
	<xsd:attributeGroup ref="ReferenceAttributes" />
</xsd:complexType>

<xsd:attributeGroup name="ReferenceAttributes">
   <xsd:attribute name="name" use="required" type="xsd:string"/>
</xsd:attributeGroup>
```

The “name” attribute in the “ReferenceType” is of the format “namespace:name” where “namespace” identifies the document where the data is declared and “name” identifies the data . Naming convention for elements of type “ReferenceType” follows a specific format i.e. “rSolid” and “rMaterial”.
The difference between a inline definition of element data or reference to an element data can be understood by following example,

```
<LogicalPart name="LogicalPart2">
	<Box name="Solid4" Dx="5" Dy="2" Dz="3"/>
</LogicalPart>
<LogicalPart name="LogicalPart3">
	<rSolid name="solids:Solid1"/>
</LogicalPart>
```

In the above example data for “Solid4” is defined inline “LogicalPart2”, but “LogicalPart3” contains reference to data defined in another document i.e. solids.
Migration software does not differentiate between inline declarations and reference types. The main reason for not resolving the references such as rSolid is to keep the migration software as generic as possible.

So if we have a complex type,

```
<xsd:complexType name="ExampleType">
	<xsd:element ref="rMaterial" />
	<xsd:element ref="Solid" />
</xsd:complexType>
```

where “rMaterial” is DDD reference type and “Solid” defines inline data. This would translate to following SQL statement,

```
CREATE TABLE A_AND_B(
```

ExampleType_ID integer PRIMARY KEY AUTO_INCREMENT,
rMaterial string,
solid string)

5.2 Data Migration

The Data migration component needs to have the information about the database table structures and complex type definitions before it can start reading the data from XML instance documents. As every complex type in XML schema documents translates into a database table, so this information is needed while migrating data from instance documents to relational database. The data migration components reads the document as a DOM tree [14] and associates each node element declaration with a database table. After associating node element types with database tables, each node element instance in the instance documents is read in as a tuple in the database table, this procedures continues untill all the elements are read in. The data Migration component does not try to resolve the DDL references [4] as they are very specific to the DDL schema and will compromise the generic approach of reading data but instead DDD core software is responsible for correctly resolving the references to build the DDD transient object model.

5.3 Schema Evolution

XML Schemas may evolve over time, introduce new types and /or change and delete the already defined types. If a new complex type is added, it can be mapped to a new database table and the data corresponding to the type can be read from the XML instance document. If a change or an update affects an already defined complex type then the database table corresponding to that type has to be altered, which might result in changing other tables depending upon the table relationships. If evolution is such that it affects the already mapped relational structure then a new migration is necessary, but if the evolution introduces new types or deletes an already defined complex type which does not have a relationship with another complex types then it will be sufficient to just add or remove a new relational table. Consider the following cases;

If a new complex type is introduced in an XML schema document,

```
<xsd:complexType name=”NewType”>
  <xsd:attribute name=”name” type=”xsd:string” use=”required”/>
  <xsd:attribute name=”value” type=”xsd:string” use=”optional”/>
</xsd:complexType>
```

then it is only necessary to introduce a new database table for the type “NewType”, without changing the already defined relational structure. After creating a relational table for a newly added complex type, the data Migration component can be used to read in the data from the XML instance documents and populate the newly introduced relational table.

If the newly introduced complex type includes data from already defined complex types,

```
<xsd:element name=”ElementaryMaterial” type=”ElementaryMaterialType”/>

<xsd:complexType name=”NewType”>
  <xsd:sequence >
          <xsd:element ref=”ElementaryMaterial” />
 </xsd:sequence >
  <xsd:attribute name=”W” type=”xsd:string” use=”required”/>
</xsd:attributeGroup>
</xsd:complexType>
```

then it will be sufficient to create a new relational table corresponding to the complex type and read in data from the XML instance documents.

If the definition of some already existing complex type changes e.g. by including additional attributes , elements or including elements of some newly defined complex type, then it will be necessary to reflect this change in the relational schema i.e.

```
<xsd:element name=”SomeElement” type=”NewType”/>

<xsd:complexType name=”MaterialType”>
  <xsd:sequence >
          <xsd:element ref=”SomeElement” />
 </xsd:sequence >
  <xsd:attributeGroup ref=”MaterialAttributes”/>
</xsd:complexType>

<xsd:attributeGroup name=”MaterialAttributes”>
  <xsd:attribute name=”name” type=”xsd:string” use=”required”/>
  <xsd:attribute name=”symbol” type=”xsd:string” use=”optional”/>
</xsd:attributeGroup>
```

This implies that the relational table for the complex type “MaterialType” along with all the other complex types derived from MaterialType have to reflect this change and in this case it will be necessary to change the schema of the already defined tables. The same rules apply for deleting already existing types, or deleting some elements and attributes of already existing types. Depending upon the effect of changes on the already defined relational schema, a fresh migration may or may not be necessary, and if required it can be performed by reading in a new XML Schema and reading the data from XML instance documents.

6 Implementation

There are number of parsers and tools that can validate XML documents, but the tools which query and model XML Schema itself are still under development. Schema Infoset library aka IBM XSD package offers an API to model concrete and abstract representation of the W3C XML Schema Language as specified in the W3C specifications. This is a useful tool for the developers who want to build schema aware applications and tools. This API is useful to model an XML Schema in terms of an Object model. For a detailed discussion and documents on this API refer to [15].

In the migration software, the Schema Infoset library is used to extract information about various types, elements and groups defined in the DDL schema. The Infoset library is used to query the types i.e. simple types and complex types in the DDL schema and find out about their attributes and elements. It also allows us to discover element, element group declarations, and attribute group declarations in the DDL schema document. After all the required information about the structure of the schema document has been extracted, it is used to perform the mapping from the XML to the relational model. Besides using classes in the Schema Infoset library, following classes and packages were implemented as part of the current project.

6.1 Implemented Classes and Packages

mapper package

The mapper package contains classes which map an XML Schema into a relational schema. It makes use of the Infoset library to gather information about the Schema document and then constructs the corresponding relational structure. It contains classes which analyze various structures in the XML schema document, complex types, simple types, attribute groups and classes to represent these structures in an application specific manner to facilitate the mapping from XML to the relational schema. Figure 6 gives an overview of the different classes in the mapper package, that is followed by a short description of the classes

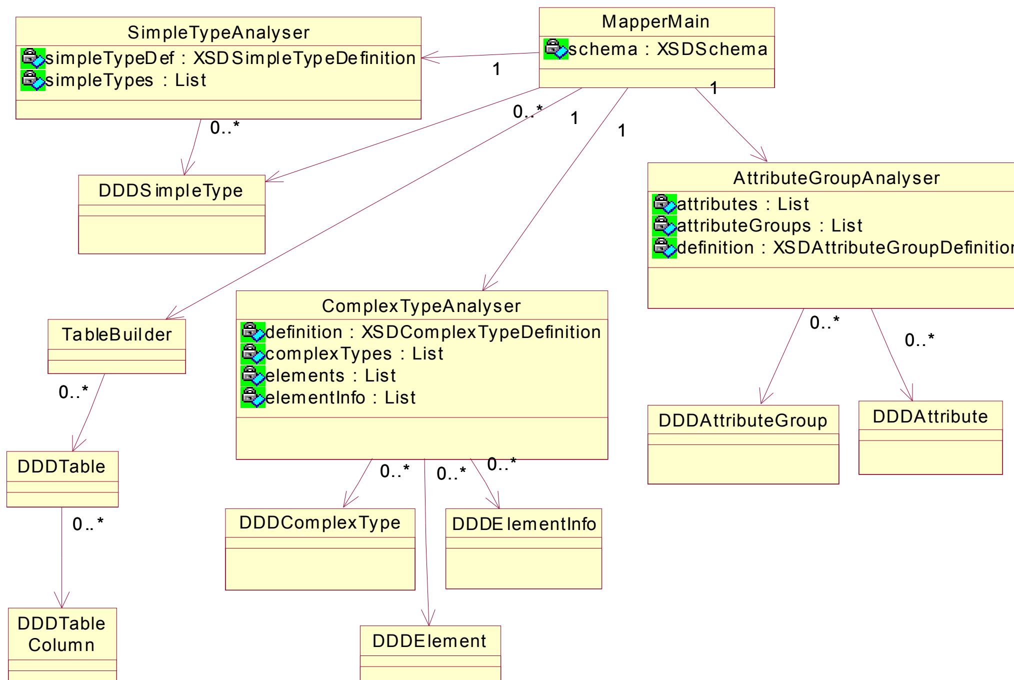


Figure 6. Class diagram of the mapper package.

MapperMain class

MapperMain class is the entry point in the mapper package. It uses the Infoset library to gather information about the schema structure and then uses the ComplexTypeAnalyser ,SimpleTypeAnalyser, and AttributeGroupAnalyser classes to investigate the schema structure and builds the information base to be used by the TableBuilder to make the database tables.

ComplexTypeAnalyser class

ComplexTypeAnalyser is passed the list of complex type definitions and it analyses them to construct the application specific complex type objects in memory which are then mapped to the relational structure. One of the important task performed in the ComplexTypeAnalyser is the construction of the ElementInfo class objects , which are used by the Reader (described in the dataReader package) to find out about the relationships among the elements and element occurrences and types associated with elements.

SimpleTypeAnalyser class

SimpleTypeAnalyser is used to gather information about the Simple type definitions in the schema documents. These simple type are represented as database table columns in the relational database system.

AttributeGroupAnalyser class

This class is used to gather information about the attribute groups from the schema document , and associate the attributes to the simple types. Currently it does not take into considerations the constraints

and patterns expressed through simple type definitions and this task is left to ODBC component of the mapping software. The AttributeGroupAnalyser class is also discovers inline simple types.

DDDTable and DDDTableColumn classes

These are the utility classes which contain the application specific format to perform the mapping. The DDDTable class does one to one mapping onto a database table and contains the DDDTableColumn objects which correspond to the database table columns. Through the DDDTableColumn class various attributes / constraints i.e. null values, data types, default values, are expressed which in turn are reflected when mapping these constructs to database table columns.

TableBuilder class

This class is used by the MapperMain to translate the gathered information about the XML schema to a relational structure. This class uses DataAccessBean and DataAccessServlet (see package util) to perform the task.

Miscellaneous classes

Utility classes like DDDSimpleType, DDDComplexType, DDDAttribute, DDDAttributeGroup and DDDElement are used to model the information about corresponding types and elements in the DDL schema document. Note that these classes do not have any relation to the DDL types and elements but are used to translate the information to a format which is easier to migrate to a relational structure. The DDDElementInfo class represents the information about elements present in the schema document and is used by the Data Migration component (see package dataReader) to discover the relationships and occurrences of the elements.

dataReader package and util package

The dataReader package contains classes responsible for reading the data in the XML instance documents and storing the data into the relational database. It uses JDOM API[16] to read the instance documents. The util package contains classes responsible for interacting with the relational database for detector description and perform various operation like making the tables, inserting and deleting the data, making selections. Figure 7 shows the classes of the reader and util package, and is followed by a short description of the package.

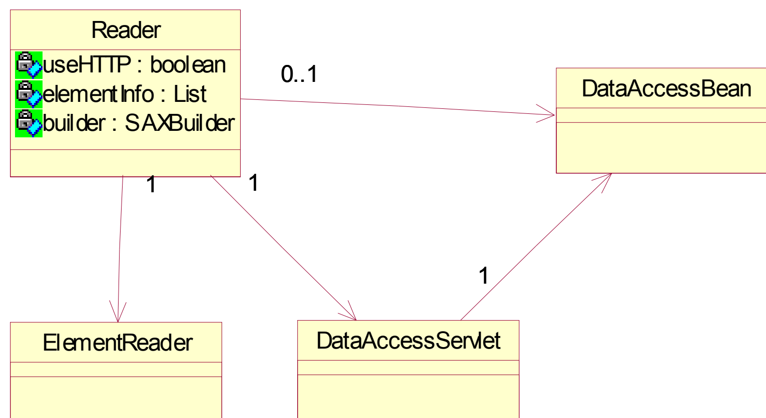


Figure 7. Class diagram of reader and util packages

dataReader.Reader class

This class is used to read the data from the instance documents and store it in the relational database. It uses the ElementReader class to obtain structural information about the schema document and reads in

the schema for the relational database. For storing the data to the database, the Reader class can either do it through DataAccessServlet or use the DataAccessBean depending on the startup configuration.

dataReader.ElementReader class

The ElementReader class is used by the Reader class to obtain information about the elements in the schema document and record their properties, maxOccurs and minOccurs, which are used by the Reader class to effectively store the data in the database.

util.DataAccessBean class

The DataAccessBean is responsible for providing the data access layer to the mapping software. It performs all the database related operations such as creating the tables, inserting and deleting the data and performing the data selection. The separation of data access layer from the rest of the application helps achieving a useful layer of abstraction where the core software does not need to worry about the database specific features. This also helps in porting the application to different databases.

util.DataAcessServlet class

This class provides a web component which can talk to DataAcessBean. It is useful in situations where a web server is used as an intermediate layer between the mapping software and the data access layer.

6.2 DDL SQL Data Model

The following paragraphs contain a description of the generated SQL data model for the detector description data translated by the migration software. The Boxes in the following figures within this section represent relational tables.

Inheritance

As discussed in the design section, an XML Schema can contain inheritance relationships. The data model for DDL schema contains the following inheritance relationships (figure 8 and 9). Inheritance between tables is implemented by copying the fields of the base table (materialType and solidType) to the sub tables.

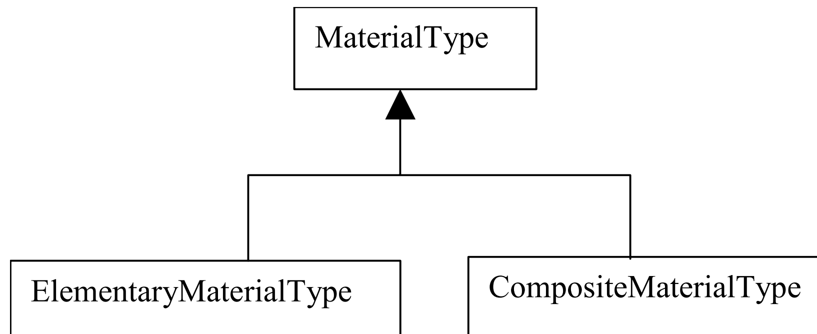


Figure 8. Inheritance relationship for material group

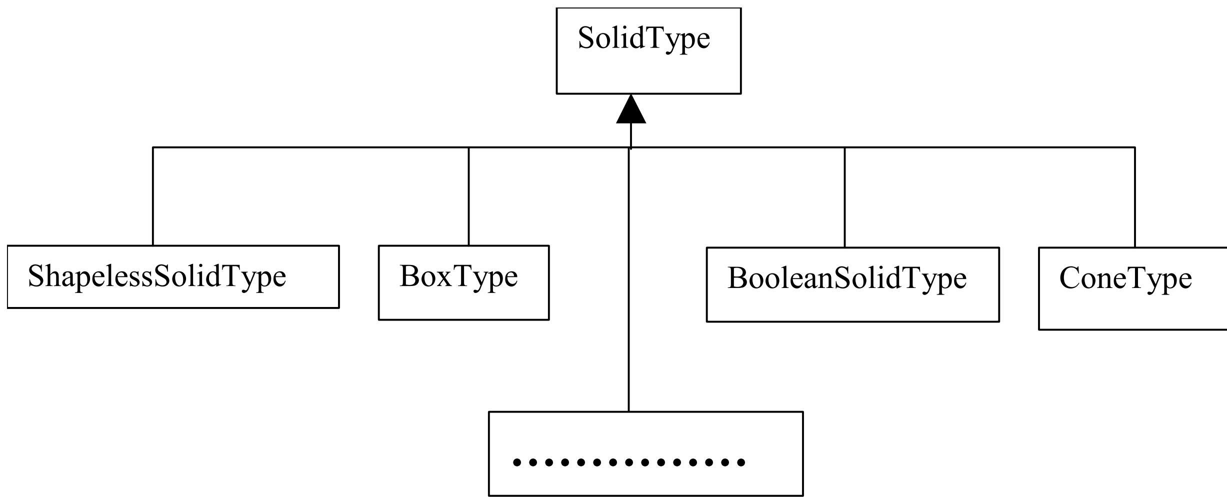


Figure 9. Inheritance relationship for solid group

Modeling relationships

In the design section, the strategies to model relationships expressed in the DDL Schema were discussed. The following paragraphs contain snap shots of the resultant relationships in the relational model for the DDL Schema.

- **One to one relationships**

One to one relationship expressed in DDL Schema is modeled in the following manner (Figure 10).

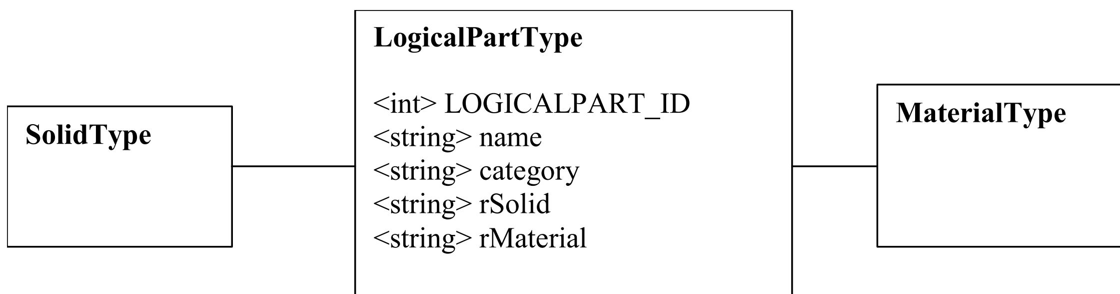


Figure 10. One to one relationships

- **One to many relationships (many to one)**

The following figure shows one to many relationship as expressed in the relational model for the DDL Schema (Figure 11).

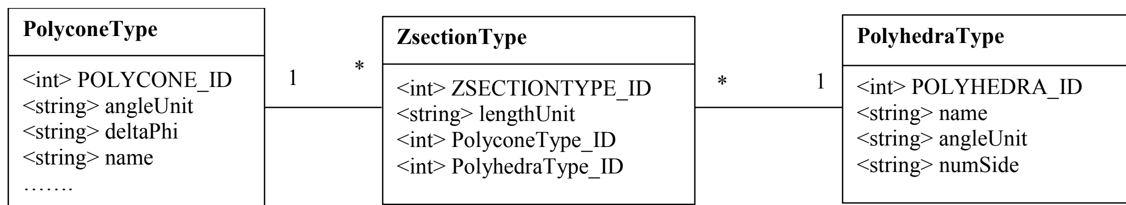


Figure 11. One to many relationship

6.3 MySql Client

A MySQL data access client has been implemented, which is used by the DDD core to build the transient object model from the relational database. Previously the only way to build transient object model was by parsing XML instance documents. The procedure to build the DDD transient object model from relational database is different from constructing the same model using XML files. The complete detector description data is present in the relational databases tables and no parsing of the description is required to get access to this data. The data access client starts to read data from one table at a time and construct the transient object model. The DDD core software does not impose any specific order for object construction and this means that objects can be constructed in any random order. For example consider the following composite material type[7];

```
<CompositeMaterial name="Water" density="1*g/cm3" symbol="" method="mixture by weight">
<MaterialFraction fraction="0.11190083">
              <rMaterial name="materials:Hydrogen" />
</MaterialFraction>
<MaterialFraction fraction="0.88809917">
              <rMaterial name="materials:Oxygen" />
  </MaterialFraction>
</CompositeMaterial>
```

This implies that the composite material object Water is made up of two elementary material objects, Hydrogen and Oxygen. In order to make a DDD Water object it is not necessary to have DDD Hydrogen or DDD Oxygen objects present in the memory. Data access client constructs DDD objects for Water independent from the DDD objects for Hydrogen and Oxygen and the DDD core software takes care of building the correct relationships and hierarchy for the transient object model.

While reading the detector description data from the relational tables string values are converted to the corresponding numeric values. This facility is provided by the DDD core software and the data access client passes the string values to the DDD core software and these values get converted to their respective types.

DDL uses xsd:string for representing a numeric value. Numeric values can also be mathematical expressions based on the parameters defines in the DDL, e.g.

```
<Translation x=`[CMSRadius]/10*sin(20*deg)`
             y=`[CMSRadius]/10*cos(20*deg)`
             z=`2*m` />
```

The DDD core software is aware of expressions and knows how to evaluate them to construct corresponding numeric values. XML documents and relational tables are required to represent expressions as string values and leave the task of expression evaluation to the DDD core software. This also enables in expressing explicit dependencies on some base numbers defined in the DDL e.g. CMSRadius.

The transient DDD object model constructed using data from the relational database generate by the migration software and using the DDD XML files was consistent with the transient object model constructed using XML files containing the same data. Because the DDD software is layered it is straightforward to "plug in" a different persistency mechanism by changing the persistency layer of the DDD software without affecting the other layers.

7 Comparing XML and Relational data model

The primary focus of this research was to demonstrate an automatic migration of DDD XML schema and data to a relational model and to see if the migrated relational model offers any comparative advantages over XML based model. However it must be remembered that the current DDD data model is optimized for XML based implementation and the relational model, obtained as a result of the migration, is tightly

[7] The density has been expressed as " density="1*g/cm3" instead of " density="1" units="g/cm3", because to our view the former is more reader friendly when editing and reading large documents

coupled to the XML model. A relational data model, designed from scratch, for DDD may be more efficient than the current XML implementation.
The implementations of the relational and XML data models are compared for efficient retrieval of data and query syntax. For the relational model an Intel Pentium4 1.6 GHz, 256 MB memory machine was used to execute queries on a locally installed MySql server version 3.23 where as the XML queries were executed on Intel P IV 1.8 GHz,256 MB memory machine using a local installation of X-Hive – a native XML database. It is important to note that the performance metrics are very specific to the database and XQuery engines used for the purpose.

7.1 Test queries

It can be seen from the following queries that XQuery is more verbose than SQL and in some cases the XQuery had a specific output format i.e. see tables 6 and 9 below. The SQL queries executed on single table or at most joined two tables to get the result whereas the execution of XQuery queries involved a set of documents.

7.1.1 Select all logical parts

The result of the SQL query, as can be seen from Table 2, is a sequence of tuples whereas the output of the XQuery query is in XML format. The XQuery in Table 1 does not enforce any format on the query result and displays the result as received from querying the documents. Another important thing to note is that querying the relational database involved executing a select query on one table i.e. logicalparttype whereas XQuery had to be executed on a set of documents.

Engine	Query Syntax	Query Execution time
SQL	select * from logicalparttype	220 ms
XQuery	let $docs := collection('/Data') for $doc in $docs return $doc//LogicalPart	355 ms

Table 1. Query syntax and execution time

<table>
<tr><th>Engine</th><th>Result</th></tr>
<tr><td>SQL</td><td>
<table>
<tr><th>LOGICAL-PART_ID</th><th>name</th><th>category</th><th>rSolid</th><th>rMaterial</th></tr>
<tr><td>1</td><td>bapixel: BarrelPixel</td><td>unspecified</td><td>bapixel: BarrelPixel</td><td>materials: Air</td></tr>
<tr><td>2</td><td>bapixel: BarrelPixel-Layer1</td><td>unspecified</td><td>bapixel: BarrelPixel-Layer1</td><td>materials: Air</td></tr>
<tr><td>........</td><td>......... . .</td><td>......... .</td><td>............ .</td><td>...............</td></tr>
</table>
</td></tr>
<tr><td>XQuery</td><td><pre>
<LogicalPart name="BarrelPixel" category="unspecified">
   <rSolid name="BarrelPixel"/>
   <rMaterial name="materials:Air"/>
</LogicalPart>
<LogicalPart name="BarrelPixelLayer1" category="unspecified">
   <rSolid name="BarrelPixelLayer1"/>
   <rMaterial name="materials:Air"/>
</LogicalPart>
....
</pre></td></tr>
</table>

Table 2. Query results

7.1.2 Select copy numbers for all Posparts

The output from the queries shown in Table 3 is a list of copy numbers for all pospart elements. The SQL query involved only one database table whereas the XQuery was executed on a set of XML documents.

Engine	Query Syntax	Query Execution time
SQL	select copyNumber from posparttype	280 ms
XQuery	let $docs := collection('/Data') for $doc in $docs return $doc//PosPart/@copyNumber	658 ms

Table 3. Query syntax and execution time

Engine	Result
SQL	**copyNumber** 0 1 ……………
XQuery	0 1 2 ……. .

Table 4. Query results

7.1.3 Select all Composite materials

The output from this query is similar to one described in Section 7.1.1.

Engine	Query Syntax	Query Execution time
SQL	Select * from compositematerialtype	80 ms
XQuery	let $docs := collection('/Data') for $doc in $docs return doc//CompositeMaterial	330 ms

Table 5. Query syntax and execution time

7.1.4 Select child and copy number for a given parent (pospart)

Depending on the requirement XQuery may have to enforce an explicit format on the query result, thus resulting in extra processing. The XQuery in table 6 had to explicitly convert results into a specific format whereas the SQL query did not involve any extra processing and presented data as received from the SQL engine.

Engine	Query Syntax	Query Execution time
SQL	Select rchild, copynumber from posparttpe where rparent='mb:MB7Y' and copynumber <10	80 ms
XQuery	let $docs := collection('/Data') for $doc in $docs for $pos in $doc//PosPart where $pos/@copyNumber <10 AND $pos//rParent/@name="mb:MB7Y" return <result> <child> { $pos/rChild/@name/string() } </child> <copyno> { $pos/@copyNumber/string() } </copyno> </result>	580 ms

Table 6. Query syntax and execution time

<table>
<tr><th>Engine</th><th>Result</th></tr>
<tr><td>SQL</td><td><table><tr><th>copyNumber</th><th>rChild</th></tr><tr><td>0</td><td>mb:MP7L</td></tr><tr><td>1</td><td>mb:MP7L</td></tr><tr><td>………… .</td><td>……………</td></tr></table></td></tr>
<tr><td>XQuery</td><td><result>
<child>mb:MP7L</child>
<copyno>0</copyno>
</result>
<result>
<child>mb:MP7L</child>
<copyno>1</copyno>
</result>
<result>
<child>mb:MP7L</child>
<copyno>2</copyno>
</result>

....</td></tr>
</table>

Table 7. Query results

7.1.5 Select translation for a pospart with given copy number

The pospart data is stored in two separate tables in the relational database system i.e. posparttype and translationtype, and was joined by the translation_id column in the posparttype table and the translation column in the translationtype table, whereas the pospart element in the XML files contains the complete data for a posparttype element. So the SQL query in table 8 involved executing a join on the posparttype and the translationtype tables whereas XQuery operated on only one element i.e. posparttype in the XML instance documents.

Engine	Query Syntax	Query Execution time
SQL	select t.x ,t.y ,t.z from translationtype as t , posparttype as p where t.translation_id=p.translation and p.rparent='mb:MB7Y' and p.copynumber=3;	50 ms
XQuery	let $docs := collection('/Data') for $doc in $docs return $doc//PosPart[rParent/@name = "mb:MB7Y" and @copyNumber = "3"]/Translation	500 ms

Table 8. Query syntax and execution time

7.1.6 Selecting the materials and fractions in a composite material

The SQL query in table 9 joined the compositemateriatype and the materialfractiontype tables for executing the query whereas the XQuery operated only on the CompositeMaterialType element in the XML instance documents. However the XQuery had to explicitly format the query result as described in section 7.1.4.

Engine	Query Syntax	Query Execution time
SQL	select mf.rmaterial,mf.fraction from compositematerialtype as c, materialfractiontype as mf where c.name=mf.compositematerialtype_id and c.name='materials:water'	50 ms

XQuery	let $docs := collection('/Data')
	for $doc in $docs
	for $mat in $doc//CompositeMaterial
	[@name=”Water” or ends-with(@name,”Water”)]
	return
	
	<result>
	{
	for $m in $mat/MaterialFraction
	return
	<Component> {
	($m/@fraction/string() , $m//@name/string())
	} </Component>
	}
	</result>
		350 ms

Table 9. Query syntax and execution time

<table>
<tr><th>Engine</th><th>Result</th></tr>
<tr><td>SQL</td><td><table><tr><th>rMaterial</th><th>fraction</th></tr><tr><td>materials:
Hydrogen</td><td>0.11190083</td></tr><tr><td>materials:
Oxygen</td><td>0.88809917</td></tr></table></td></tr>
<tr><td>XQuery</td><td><result>
<Component>0.11190083 materials:Hydrogen
</Component>
<Component>0.88809917 materials:Oxygen
</Component>
</result></td></tr>
</table>

Table 10. Query result

Despite the fact that the DDL schema is optimized for the XML format the comparative query response time for relational model was a factor 1.6 to 10 times faster than that of XML model. The relational database engine was also running on a slightly slower machine.

7.2 Comparing XML and Relational Database Size

The size of the database both in the XML format and in a relational format are now compared. This indicates the constant overhead involved in the use of a relational database system. Table 11 shows the different sizes of the databases. Microsoft Access 2000 and MySql were used for this testing.

XML Files	520 KB	1 MB	2 MB	4.2 MB
Access 2000	1.28 MB	1.84 MB	3.32 MB	6.47 MB
MySql	695 KB	1.02 MB	1.80 MB	3.50 MB

Table 11. Data size

The overhead in Access and MySql databases is due to the fact that these database files also contain data for efficient data retrieval. For the large data size, the size of the MySql database is smaller than the XML file size. The reason for this smaller size is that the XML files contain data and their descriptions (XML tags) where as a relational database stores the data only. The schema describes the data within a relational database and it does not have to be “human readable” as with XML files, and can be stored much more efficiently than XML tags. Interestingly enough the size of Access database is twice then that of MySql database, mainly due to the fact that MySql uses highly compact table format i.e. MyISAM [17] to store the database. MyISAM databases are very compact on disk.

8 Related Work

A number of products have been developed for XML migration and efficient management of XML documents. A strategy is to migrate the data to a relational database. But it is also possible to store the data in different data sources with different persistency strategies such as object-oriented databases. Several projects have explored migration to relational databases. The advantage of this approach is that it is possible to query XML documents using SQL query engines. SQL query engines are still faster in executing and retrieving data than the XML query engines.

Database vendors such as IBM and ORACLE offer modules to store XML data in a relational database. Schmidt et. al[18] describe how XML documents can be stored in a relational database by fragmenting the document into nodes and attributes. Bohannon et. al[19] discusses another approach to store XML documents in a relational database. XML documents are also fragmented into nodes. But different tables are created for the different nodes (different in name). Mani and Lee [20] describe transformations from XML to a relational database based on regular tree grammars. Mani and Lee identify several XML schema constructs that are difficult to transform into a relational schema. Several techniques are discussed to handle these XML schema constructs and to simplify the relational schema, [20] uses a similar approach as [19] but focuses also on optimization of the relational schema.

The approaches discussed above are the basis for XML databases. The added value of XML databases is that they provide an XML interface on top of the backend that stores the XML files. Once XML files are stored in a relational database they can be queried using SQL constructs. XML databases would provide an interface on top of a SQL engine that would allow applications and users to query and navigate using Xpath and XQuery . The difference between the different XML database products are (amongst others):

- How is the XML stored in the underlying database?
- What database is used (relational, object-oriented)?
- Indexing techniques on the XML data.
- How complete is the XML interface to the XML files?

Numerous XML database products have been developed but essentially they deal with storing XML documents and serve more like frameworks for storing XML documents and do not deal with mapping between the XML schema and the relational schema.

The following is a brief description of few of them:

Ipedo XML Database [21] stores well-formed XML documents and provides access to both whole documents and document fragments. The database provides access by an object cache, indexes, a query and transformation engine, and a mechanism to make persistent the parsed state of the document. Documents are organized into collections. Indeces are maintained on the collection level, and can be created for specific elements or attributes. Users write queries in XPath, which has been extended to allow queries across multiple documents in a collection.

Lore [22] is a database designed for storing semi-structured data. Although it predates XML, it has recently been migrated for use as an XML database. It includes a query language (Lorel), multiple indexing techniques, a cost-based query optimizer, multi-user support, logging, and recovery, as well as the ability to import external data. Because Lore is designed for use with semi-structured data, XML documents without DTDs can be easily stored.

Xindice [23] is a native XML database written in Java that is designed to store large numbers of small XML documents. It can index element and attribute values and compresses documents to save space. Documents are arranged into a hierarchy of collections and can be queried with XPath. (Collection names can be used as part the XPath query syntax, meaning it is possible to perform XPath queries across documents.)

X-Hive is a native XML database for XML data processing and storage. It provides support for XML 1.0, XQuery , XPath and related standards. Queries can be executed on single documents as well as on set of documents i.e. collections. Other features include version management, indexing, link management, publishing and schema verification.

Tamino [24] is a suite of products built in three layers: core services, enabling services, and solutions (third-party applications). Core services include a native XML database, an integrated relational database, schema services, security, administration tools, and Tamino X-Tension, a service that allows users to write extensions that customize server functionality.

There are a number of projects looking at mapping XML to an Object Model , these projects perform a mapping between XML – Object Model – relational Model . But the mapping from object model to the relational model is equivalent to storing objects in relational databases and is not a true transformation from the object model to a relational model. Few of the interesting projects are:

The Java ™ Architecture for XML Binding (JAXB) [25][26] provides a fast, convenient way to create a two-way mapping between XML documents and Java objects. JAXB is a java XML manipulation solution based on existing technologies like Simple API for XML (SAX) [27] and Document Object Model (DOM) [14]. Given a schema, which specifies the structure of XML data, the JAXB compiler generates a set of Java classes containing all the code to parse XML documents based on the schema.

Castor XML is an XML data-binding framework [28]. Unlike the two main XML APIs, DOM (Document Object Model) and SAX (Simple API for XML), which deal with the structure of an XML document, Castor enables one to deal with the data defined in an XML document through an object model, which represents that data. Castor XML can marshal almost any "bean-like" Java Object to and from XML.
The source generator component in Castor is a complete XML data-binding framework that offers the ability to generate Java code from an XML Schema.

LegoDB is a cost-based XML storage mapping engine that automatically explores a space of possible XML to relational mappings and selects the best mapping for a given application. LegoDB offers facilities for automatic and application driven mapping. It has not been released as of writing this note but promises a very flexible and adaptive mapping strategy from XML to relational format.

9 Conclusion

This document discussed the migration from the XML detector description data to a relational model maintaining the semantic structure of the data. It involves migrating the XML Schema to a corresponding relational schema and reading the data from XML instance documents to populate the relational database. The migration does not assume any specific schema format and is independent of the DDL Schema [10] .It can also handle schema evolution by either introducing new tables, if new “types” are introduced or by changing the already existing tables if the structure of corresponding “types” is altered.

Currently XML is being used at CMS as a mechanism to describe the detector in a consistent and coherent manner and to serve as a common source of information for CMS offline software [10][11]. Both XML files and relational databases offer advantages for the management and manipulation of Detector Description data [10]. XML files can serve as a mechanism for physicists to insert or edit detector description data and can serve as effective data exchange format, where as relational databases serve as a mechanism for efficient data storage, retrieval and management thus offering the “best of both worlds”.

The primary focus of this research was to demonstrate an automatic migration of DDD XML schema and data to a relational model and to see if the migrated relational model offers any comparative advantages over XML based model. However it must be remembered that the current DDD data model is optimized for XML based implementation and the relational model, obtained as a result of the migration, is tightly coupled to the XML model. It is possible to design a relational schema, from the scratch, for DDD which is more efficient than current XML based model. Thus this work can not be the basis for establishing the efficiency of XML and relational based DDD models but can be looked as a tool which can improve interoperability between XML and relational DDD implementations.

This is very initial work and the Schema Migration component handles only global entities defined in the DDL schema. Further work can be undertaken to migrate the complete XML Schema structure to a relational model.

Annex A: Brief description of DDL numeric types

In DDL schema the numeric quantities are expressed as an extension to schema built-in type ‘string’, instead of float or integer. The reason of having xsd:string instead of numerical types in the XML schema is that the values not necessarily represent 'naked' numbers, but expressions made out of user-defined constants, mathematical functions (sin, cos, ...) and operators (+,-,/,...). Evaluating such expression is beyond XML schema's capabilities. However, for all these string-typed expressions, the schema foresees further attributes, allowing the document processor to infer the correct types, i.e. physical quantities and units.

For example: the definition of the attributes of the Tubs-solid looks like this:

```
<xsd:attributeGroup name= TubsAttributes >
  <xsd:attribute name='rMin'  type='lengthT'  use='required' />
  <xsd:attribute name='rMax'  type='lengthT' use='required'/>
  <xsd:attribute name='dz'  type='lengthT' use='required'/>
  <xsd:attribute name='startPhi'  type='angleT'  use='required'/>
  <xsd:attribute name='deltaPhi'  type='angleT' use='required'/>
  <xsd:attribute name='lengthUnit'  type='lengthUnitT' default='mm'  use='optional'/>
  <xsd:attribute name='angleUnit'  type='angleUnitT' default='deg' use='optional'/>
</xsd:attributeGroup>
```

And;

```
<xsd:simpleType name="lengthT">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>

<xsd:simpleType name="lengthUnitT">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="m"/>
    <xsd:enumeration value="cm"/>
    <xsd:enumeration value="mm"/>
    ....
  </xsd:restriction>
</xsd:simpleType>
```

The attribute 'rMin', for example, has type 'lengthT' - which is a string. But the document processor can use the typename (lengthT) to infer the physical quantity - a length. Altogether, this provides sufficient information for a document processor to evaluate the expression-string to a numerical value in the corresponding physical units. Such a processor is NOT a standard XML parser, but an application built on it extending the standard XML capabilities - such as our DDD software.

References

[1] Extensible Markup Language XML, http://www.w3.org/XML

[2] Korth H, Silberschatz A, Database System Concepts

[3] Christian Arnault, Stan Bentvelsen, Steven Goldfarb, Marc Virchaux, Christopher Lester, “A generic approach to the detector description in Atlas”, CHEP 2000

[4] M. Liendl, F. van Lingen, M. Case, T. Todorov, P. Arce, A. Furtjens, V. Innocente, A. de Roeck, "The Role of XML in the CMS Detector Description", in proceedings of CHEP 2001

[5] Frank van Lingen, Martin Liendl, Michael Case, “Detector Description Domain Architecture and Data model “, CMS note 2001/057

[6] Nassem Bhatti, Jean-Marie Le Goff, Waseem Hassan, Zsolt Kovacs, Peter Martin, Richard McClatchey, Heinz Stockinger, Ian Willers, “Object Serialisation and Deserialisation Using XML” , 8th International Conference on Advanced Computing

[7] G. Kappel, E. Kapsammer, W. Retschitzegger, “XML and Relational Database Systems - A Comparison of Concepts”

[8] Scott W. Ambler, “Mapping objects to relational databases”

[9] Xquery, http://www.w3.org/XML/Query

[10] Xpath, http://www.w3.org/TR/xpath

[11] XML frameworks, http://xml.apache.org/xindice, http://exist.sourceforge.net/index.html

[12] X-Hive native XML database, http://www.x-hive.com

[13] XMLSchema, http://www.w3.org/XML/Schema

[14] Document Object Model, http://www.w3.org/DOM/

[15] IBM Infoset Library, www.eclipse.org/xsd

[16] JDOM API, www.jdom.org

[17] MySQL manual, MySQL Table Types , http://www.mysql.com/doc/en/Table_types.html

[18] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, Florian Waas, “Efficient Relational Storage and Retrieval of XML Documents” Lecture Notes in Computer Science 2000

[19] Philip Bohannon, Juliana Freire, Prasan Roy, Jayant R. Haritsa, Jerome Simeon, Maya Ramanath, “LegoDB:Customizing Relational Storage for XML Documents” Proceedings of the 28th VLDB Conference Hong Kong, China, 2002

[20] Murali Mani, Dongwon Lee, “XML to Relational Conversion using Theory of Regular Tree Grammars” Proceedings of the 28th VLDB Conference Hong Kong, China, 2002

[21] Ipido, http://www.ipido.com

[22] R. Goldman, J. McHugh, J. Widom, “From Semistructured Data to XML: Migrating the Lore Data Model and Query Language” Proceedings of the 2nd International Workshop on the Web and Databases (WEBDB99) Philadelphia, Pennsylvania, June 1999

[23] Xindice, http://www.apache.org/xindice

[24] Tamino, http://www.softwareag.com/tamino

[25] Java Platform, http://www.java.sun.com

[26] Java TM Architecture for XML Binding (JAXB), www.java.sun.com/xml/jaxb

[27] Simple API for XML, http://www.saxproject.org/

[28] Castor Project, www.castor.exolab.org