Uppsala Master's Theses in
Computing Science no. 260
Examensarbete DV3 20 p
2004-01-12
ISSN 1100-1836

Importing XML Schema into an Object-Oriented Database Mediator System

BY

Tony Johansson Richard Heggbrenna

November 2003

Information Technology Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Supervisor: professor Tore Risch
Examiner: professor Tore Risch

Passed:

Importing XML Schema into an Object-Oriented Database Mediator System

Master's Thesis in Computer Science 20 p
Uppsala University

BY

Tony Johansson
tojo3423@student.uu.se
Uppsala University

Richard Heggbrenna
rihe8139@student.uu.se
Uppsala University

November 2003

This work was carried out at
Uppsala Database Laboratory (UDBL)
Uppsala University
PO Box 513
SE-751 20 Uppsala
Sweden

Abstract:

A *mediator system* is a middleware database system that provides uniform queries and views over several different back-end heterogeneous data sources. An object-oriented mediator system can help solve integration problems between *eXtensive Markup Language* (XML) instance documents and different software applications. It is important to integrate XML with databases as XML provides a standard technique to describe data in files. The thesis describes an implemented prototype system for importing XML Schema definitions into an object-oriented mediator system. The mediator system provides an object-oriented query language to specify queries and views over combinations of data from XML documents, relational databases, and other kinds of data sources used by applications. Adaptive translation rules and mapping of datatypes allow automatic importation of XML Schema definitions into the object-oriented mediator system. As more XML Schema definitions are imported, the mediator's schema is dynamically extended using object-oriented data definition statements. This requires the mediator system to be capable of dynamically extending and modifying its schema.
Keywords: XML Schema, XML, Mediator, Object-Oriented databases

Table of contents

Figures and tables

Chapter 1

Introduction

An increasingly popular way to represent data is using the *eXtensible Markup Language* (XML) [1], which is a markup language to express structured data in files. These XML files are commonly called *XML instance documents*. Since XML is a markup language with no pre-defined tags, in contrast to for example *HyperText Markup Language* (HTML) [2] that contains many pre-defined tags, it is possible to declare any kind of descriptive tags for the contained data. Hence, to exchange XML instance documents in a meaningful way requires their internal data to be described so that the various parties involved will interpret them correctly and consistently.

A *schema* is a description of data in a database, often also referred to as *meta-data*. A *data model* is a language that is used to define schemas, for example the *XML Schema definition language* [4][5] provides a data model for defining schemas of XML instance documents. To avoid confusion we make a distinction between the terms *XML Schema definition language* (a data model) and an *XML Schema definition* (a schema expressed using the XML Schema definition language) [3].

An XML Schema definition contains rules governing the structure and format of an XML instance document. It also describes, at least informally and often implicitly, the intended conceptual meaning of an XML instance document's components. An XML Schema definition is, in other words, a specification of the syntax and semantics of a potentially infinite set of XML instance documents [15]. XML Schema definitions allow different involved parties to share the documents between each other in a meaningful way and an XML Schema definition helps the parties to interpret the XML instance documents correctly and consistently.

The internal structure and datatypes of the XML Schema definition language has several features beyond what the *XML 1.0-Document Type Definitions* (DTD) [1] definition language provide [3]. The main difference between an XML Schema definition and a DTD definition is that XML Schema definitions include extensive support for datatyping.

To help solve the problem of integrating and querying data from many different sources, the *wrapper-mediator* approach provides intermediate virtual databases, called *mediators*, between different kinds of data and the applications using them. A w*rapper* is an interface that translates a data source's data model to a common data model known by the mediator [7].

This thesis focuses on the ability to wrap an XML Schema definition in an object-oriented database mediator system to help solve the integration problems between XML instance documents and other applications that are not using XML. Instead of creating program modules that explicitly read XML instance documents, a mediator database system provides the application with data contained inside XML instance documents by using its query language.

We have developed a schema importation prototype tool for XML Schema definitions called *Amos II XML Schema import tool* (AXSI). AXSI allows XML Schema definitions to be imported into a mediator system. Our mediator system is object-oriented and uses an object-oriented query language. When an XML Schema definition is imported the system creates an object-oriented database view in the mediator over data that is represented in XML documents described by the imported XML Schema definition. This requires that the object-oriented database mediator system understands the XML Schema definition language. Given a specific XML Schema definition, the schema importer needs to know how the XML Schema definitions are translated into object-oriented schema definitions in the mediator. In order to understand an XML Schema definition, the object-oriented database mediator system must know the semantics of the XML Schema definition language. When a schema is imported another data importation tool being developed will make it possible to also import XML documents described by the imported schema. Finally given that both XML Schema definitions and XML documents are imported the object-oriented query language of the mediator can be used to query the imported data.

The mediator system also provides the possibility to use an object-oriented query language to query and create views over several different data sources, for instance data in a relational database can be combined with data represented in XML described by an XML Schema definition. A given set of translation rules from XML Schema to object schema definitions govern the translations that take place in the wrapper.

With the schema importation tool it is possible to access from the mediator database any XML instance document conforming to the imported XML Schema definition just by using the imported object-oriented schema definitions in the mediator system. In order for this to work, the object-oriented database mediator knows a significant subset of the XML Schema definition language when performing the importation.

1.1. Questions at issue

The question is now how an XML Schema definition is imported into an object-oriented database mediator system by using such a schema importation tool. The question is divided into three sub questions:

Q1. How is a schema importation tool that imports an XML Schema definition into the object-oriented database mediator designed? If the tool is automatic, it can translate different XML Schema definitions automatically. Consequently, an analysis of several translations will reveal if the tool performs the importation correctly or not.

Q2. Can the tool translate the structures of an XML Schema definition to the database schema, in an obvious and useful way, which reflects the intended meaning of the XML Schema definition? *XML Schema part 1: Structures recommendation* shows the structure of the XML Schema definition language and the language contains several different components that govern its structure [4].

Q3. Can the tool map XML Schema definition language datatypes to corresponding datatypes in the object-oriented database mediator? A study of *XML Schema part 2: Datatypes recommendation* describes supported datatypes for the XML Schema definition language [5].

1.2. Method used

First a literature study was made about the concepts of object-oriented databases, mediators, XML, and the XML Schema definition language.

Based on the literature study a prototype XML Schema importation system was implemented. The automated tool implements the translation of structures and mapping of datatypes between the XML Schema definition language and the object-oriented database mediator data model. Importing several XML Schema definitions confirmed the correctness of the translations and mappings.

For the evaluation of the prototype the XBench benchmark [12] was used in order to validate and enhance the results of the tool. This showed that it is possible to import XML instance documents described by XML Schema definitions from XBench. The schemas used in XBench are not object-oriented and therefore a complementing object-oriented XML Schema was also developed and imported.

1.3. Presuppositions

XBench [12] is a family of benchmarks for XML databases. The benchmark suite generates XML instance documents of varying size that conform to an included set of XML Schema definitions. The included XML Schema definitions provide a sufficient subset of the XML Schema definition language to base the translations on, and, since the XML Schema definition language is rather extensive, the thesis is limited to presenting translations that works on these XML Schema definitions. The schema importation tool was considered complete when it could import these XML Schema definitions into the database mediator system. The result of this thesis however, should provide a basis for further research in which added translations provide support for a larger subset of the XML Schema definition language and as a result, an improved tool can import a wider range of XML Schema definitions into an object-oriented database mediator system.

1.4. Report overview

The arrangement of the thesis is as follows: Chapter 2 presents related work. Chapter 3 presents the background to understand this thesis. Chapter 4 describes the realization of the XML Schema definition importer tool. Chapter 5 describes the XML Schema definition importer tool. Chapter 6 is a discussion regarding this thesis and finally, Chapter 7 concludes the work and proposes future work.

Chapter 2

Related work

A system that integrates XML with applications typically do this by mapping XML documents to something else, for instance into classes in an object-oriented programming language or tables in a database. Below is a description of some interesting technologies and frameworks that relate to this thesis.

The *Java Web Services Developer Pack* (Java WSDP[1]) [13] created by SUN Microsystems is a free integrated toolkit that allows Java developers to build and test XML applications with up-to-date Web services technologies and standards implementations. The Java WSDP includes the *Java APIs for XML Processing* (JAXP) [14] and *Java Architecture for XML Binding* (JAXB) [15] technologies among others.

The JAXP framework supports processing of XML documents using the *Document Object Model* (DOM) [16] defined by W3C, *Simple API for XML* (SAX) [17] defined by saxproject.org and *XML Stylesheet Language for Transformations* (XSLT) [18] defined by W3C. This framework enables applications to parse and transform XML documents independent of a particular XML processing implementation. Depending on the needs of the application, developers have the flexibility to swap between XML processors (such as high performance versus memory conservative parsers) without making application code changes. This technology can be used to XML-enable the prototype importer tool for the database mediator system [13].

The JAXB framework provides API, tools, and a framework that automates the mappings between an XML Schema definition and Java-level binding objects. The objects are classes that can be compiled and later instantiated using an XML instance document conforming to the XML Schema definition. The framework also allows a developer to specify application specific details about how the mapping should be performed [13]. The specification of the framework is the most interesting part and not the implemented parts of the framework. The specification shows mappings for the conceptual level of an XML Schema definition, how more intricate parts of an XML Schema definition map to classes and inheritance, how XML Schema definition primitive types map to properties in those classes and so forth.

The latest release of Oracle XML DB [19] uses XML Schema definitions to allow simple element contents and attribute values to be stored in SQL columns declared to have an SQL type most similar to the corresponding XML Schema type. Many datatypes, such as sequences, can however not be mapped but are instead represented as text strings.

R. Bourret et al. has developed a database product called XML-DBMS [22] [20]. It is a middleware for transferring data in XML instance documents to and from relational databases. XML-DBMS uses an XML-based mapping language to explicitly specify transformation rules from an XML instance document to a relational database schema i.e. how classes map to tables and properties map to columns in the tables. If the XML instance documents are described by DTD definitions the transformation rules of the XML instance documents can be created automatically rather than explicitly. Currently however, XML-DBMS cannot automatically create transformation rules of XML instance documents described by XML Schema definitions and it has no transformation rules to object-oriented database schemas. By contrast our prototype tool imports an XML Schema definition as an object-oriented database view over data that is represented in XML instance documents described by the imported XML Schema definition.

[1] http://java.sun.com/webservices/

H. Lin et al. presented a framework for querying XML data through an object-oriented mediator using an object-oriented query language. The framework, called AmosXML, defines a set of translation rules that automatically generate a database schema in the mediator database from a DTD definition of an XML document, if available. If the framework reads XML documents with no specified DTD definition or if the DTD definition is incomplete the framework extends the database schema from the XML instance document structure. The framework also provides mechanisms to refine the database schema dynamically while reading XML instance documents [23].

The main differences between H. Lin et al.'s work and this thesis are that different schema definition languages are used. The AmosXML framework uses DTD definition language and in this thesis, the definition language is the XML Schema definition language. The described translation rules by H. Lin et al. can help solve future problems however, and provide much guidance in the progress of this work.

Chapter 3

Background

The following chapter is an explanation of this thesis's related concepts. It served as a resource of information for later chapters and readers are advised not skip this section unless they are already familiar with the concepts that are used in later chapters.

3.1. Databases

A database is a collection of related data and a database management system (DBMS) is a collection of programs that enables users to create and maintain a database. DBMSs can be classified according to several criteria where the main classification is based on the data model[2]. The relational data model is the main data model used in many current commercial DBMSs. The object data model is not as wide spread commercially as the relational data model, although there are some commercial implementations. Many of the relational DBMSs have been incorporating many of the concepts from the object data model. This has formed a new class of data model, the *object-relational* model. Other data models are *hierarchical* and *network* data models.

In a *relational model*, a relation can be seen as a table of values where each row represents a collection of related data. The table and column names help to interpret the meaning of the data in each row. A row is called a *tuple* and a column header is called an a*ttribute*. A relational schema is made up by a relational name and a list of attributes. The relation of a relational schema is made up by a set of *tuples* where each *tuple* is an ordered list of values.

Object oriented databases (OODB) addresses the need to be able to model more complex structured object, and the need to being able to define nonstandard application-specific operations. Those kinds of needs can be found in databases for CAD/CAM and geographic information systems (GIS). A key feature of object-oriented databases is the ability for the database designer to create objects with desired structure of complex objects and the operations that can be applied on them. In an OODB the information is stored as objects that correspond to real-world object. The objects are identified with a unique system generated *object identifier* (OID). The OIDs can be compared to the primary keys in the relational model. The main difference though, is that the OID are managed by the object-oriented DBMS rather than as user defined keys.

3.2. Query languages

The Structured Query Language SQL

SQL is the standard query language for commercial relational database systems. It provides a high-level declarative language interface, so the user only has to specify *what* the result is to consist of. The DBMS takes care of the optimization of the query and decides how to execute the query in the most efficient way. SQL has statements for data definition, query and update, and other facilities such as being able to define integrity constraints, views, security and authorization control, and transaction controls. The SQL language is based mainly on *tuple relational calculus* and it borrows some of the features

[2] A collection of concepts that can be used to describe the data types, relationship and constraints that should hold for the data in a database.

from relational algebra. SQL uses a *select-from-where* construct for queries. The ANSI[3] and ISO[4] standard organizations both form the standard organ for SQL. The current version of the SQL standard is SQL-99.

The Object Query Language OQL

The *Object Data Management Group* (ODMG) has proposed a query language, Object Query Language (OQL), for their object model [24]. The query language has close bindings to the common programming languages that have object oriented features such as C++, Java, and Smalltalk. The syntax for QOL is similar to SQL with additional object oriented features such as object identity, inheritance, polymorphism, and relationships, etc. The lack of a standard of the object data model has imposed problems with interoperability and portability. The creation of the standard for the ODMG's object data model, hopes to solve these problems. [25]

3.3. Mediators

Wiederhold originally proposes the *mediator* approach in [26]. Wiederhold offers a definition on what a mediator is: "A mediator is a software module that exploits encoded knowledge about certain sets of subsets of data to create information for a higher layer of applications". A mediator hides the complexities of different data sources by making it appear to the user that the mediator database contains all the data, when in fact the mediator itself is a virtual database connected to the different backend data sources, hence making it transparent to the user. An alternative to a mediator system is the *data warehouse* approach where information is regularly extracted from the heterogeneous data sources and loaded into a central large database called a data warehouse [27].

3.4. Amos II system

Amos II (*Active Mediator Object System*) is an object-oriented mediator system developed at the Uppsala Database laboratory (UDBL[5]) at Uppsala University in Sweden. The purpose of the Amos II project is according to [7] to *"develop and demonstrate a mediator architecture for supporting information systems where applications and users combine and analyze data from many different data sources."* Amos II consists of a mediator database engine that can process and execute queries over data stored locally or data scattered over several external data sources. Applications can access data from distributed heterogeneous sources through one or several Amos II mediators. With performance in mind, the core of Amos II has been designed as a main-memory DBMS with a data manager optimized for main-memory access.

Amos II access data from external sources through *wrappers*. A *wrapper* is a program module in Amos II that has particular facilities for query processing and translation of data from a particular class of external data sources. A wrapper consists of an interface to the external data source and a translation mechanism for translating queries in AmosQL into function calls to the interface. [28]

3.4.1. AmosQL

AmosQL is an object-oriented query language based on OSQL [29] and DAPLEX [30] with extensions of mediation primitives, multi-directional foreign functions, late binding, and active rules etcetera. AmosQL is similar to the object-oriented parts of SQL-99. Like SQL, AmosQL uses the *select-from-where* construct for queries. AmosQL is

[3] American National Standard Institute.
[4] International Standards Organization.
[5] www.docs.uu.se/~udbl

relationally complete which means that it can express all the queries that can be expressed in relational algebra. AmosQL is a combination of a Data Definition Language (DDL), Query language and a Data manipulation Language (DML). [8] [9]

3.4.2. Amos II data model

The data model of AMOS II is an object-oriented extension of a functional data model called Daplex [30]. The basic concepts of the AMOS II data model are objects, types, and functions, see Figure 1 below.

Objects

All entities in an Amos II database are modeled as objects. Both the user defined objects and the system objects are managed by the system. There are two main kinds of representations of objects: *literals* and *surrogates*. A literal is a type that is a self-described, system maintained object that does not have associated explicit object identifier (OID), e.g. numbers and strings. Literal objects can also be collections of other objects. The Amos II system supports two types of collections: 1-dimensional arrays (vector) and unordered sets with duplicates (bag). Surrogate objects are characterized by having OIDs, which are explicitly created and deleted by the system or the user. Surrogate objects are used to represent real-world entities such persons. Surrogate objects are also used to represent meta-objects such as functions as types. Literal and surrogate objects persist in the database as long as they are referenced by any other object or by external systems. Unreferenced objects are removed by an automatic garbage collector [28].

Datatypes

The Objects in Amos II are classified into types. All objects are instances of some types and the set of all instances of a particular type is called the extent of that type. The types are organized in a multiple inheritance, supertype/subtype hierarchy. If an object is an instance of a type, then it is also an instance of all the supertypes of that type; equally, the extent of a type is a subset of the extent of a supertype of that type. A type that is multiple inherited from other types has an extent that is the intersection of the extents of its supertypes [8].

The Amos II data model provides four kinds of types[6]: *stored*, *derived*, *proxy* and *integration union* types. The stored type is the regular type that is created by the *create type* statement and have its extent stored locally in the database. The instances of a stored object are maintained by the user [28].

Amos II has a system type hierarchy that consists of meta-types. A stored type is an implicitly created as a sub-type to the meta-type *UserObject*. The general syntax for creating a new stored type with AmosQL is:

```
create type <identifier>;
```

A stored type that is a subtype of another already defined type (supertype) is created with the following syntax in AmosQL:

```
create type <identifier> under <supertype>;
```

where <*supertypes*> can be a comma-separated list of types, denoting multiple inheritances. All supertypes have to be defined before their subtypes can be defined. [9]

[6] Only the stored datatype will be dealt with in this document. For information about the other datatypes, the reader is referred to section 4 in. [28]

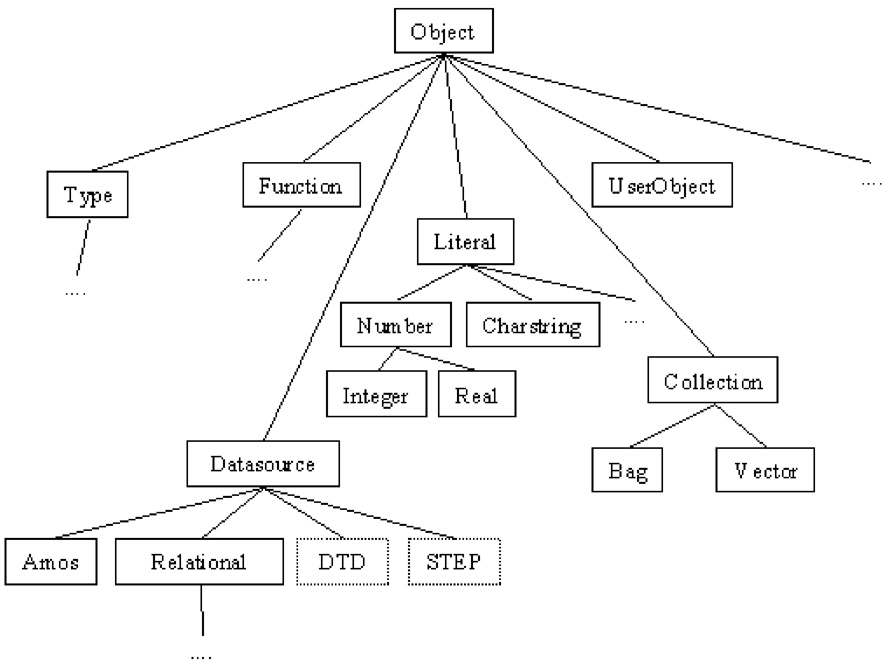

Figure 1: Part of the system type hierarchy in Amos II. [8]

Numeric datatypes

Numeric datatypes in Amos II are instances either of the system types *Integer* or *Real*, see Figure 1 above. The *Integer* and *Real* meta-types are sub-types of the system type *Number*. All numeric datatypes are sub-types of *Literal*. The *Integer* datatype can contain a signed 32 bits wide value. The floating point datatype in Amos II is a double-precision real number, see Figure 2 below.

Amos II literals:	Min Value	Max value
Integer (32 bits)	-2147483648	2147483647
Real	IEEE double-precision 64-bit floating point type	

Figure 2: Numeric datatypes in Amos II.

String datatypes

There is one string datatype in Amos II, *Charstring*. It is implemented as an array of bytes; therefore it can store an arbitrary long sequence of bytes.

Time and date datatypes

Amos II supports three data types for time and date: *Time*, *Timeval*, and *Date* [9].

- *Timeval* - is for specifying absolute time points including year, month, and time-of-day. *Timeval* has the properties: *time, date, second, minute, hour, day, month* and *year*.
- *Date* - specifies just year and date. The dates that can be represented by the datatype *Date* have to be in the interval: *1970-01-01< date < 2038-01-20. Date* has the properties *year, month* and *day*.
- *Time* - specifies time of day and has the properties *second, minute* and *hour*.

A limitation is that the internal operating system representation is used for representing *Timeval* values, which means that one cannot specify a date outside the interval: *1970-01-01< date < 2038-01-20*.

3.4.3. Functions

Functions model object attributes, methods, queries, and relationships. Functions are instances of the meta-type function. The functions can be classified into different categories depending on their implementation: *stored*-, *derived*-, *foreign*- and *proxy* functions, and *database* procedures. Functions in Amos II can be overloaded i.e. have different implementations (*resolvents*) for functions with the same name but with different arguments [28].

A function consists of a *signature* and an *implementation*. The signature defines the type(s), the optional name(s) of the argument(s), and the result of a function. For example, to model an attribute "*name*" of an object "*person*" the following *stored function* is defined using AmosQL:

```
create function name(Person)->Charstring as stored;
```

The *implementation* specifies how to compute the result of a function given the argument values [8].

Stored function

Stored functions represent properties (attributes) of objects stored in the database. Stored functions represent data stored in a database as tables in a relational database do. Stored functions can be used to model relationship between objects [8].

Derived function

Derived functions are functions defined in terms of object-oriented queries over other AMOSQL functions. Derived functions cannot have any side effects. When a derived function has been defined, a query optimizer optimizes its definition. Derived functions correspond to side-effect free methods in object-oriented models and views in relational databases [28].

Example in AmosQL:

```
create function age(Person p)->Integer as
select current_year() -  born(p);
```

Foreign functions

Foreign functions are implemented in the programming languages Java, Lisp, or C and can be used to extend Amos II with new datatypes and functionalities. Foreign functions can be *multidirectional*, which means that the system is able to inversely compute one or several argument values if the expected result value is known. Foreign functions make it possible for Amos II to access external system, where they can for example manipulate and update data structures. Foreign functions are realized through the *callout* interface and must be side effect free if used in queries. If a foreign function has a side effect, it should be used as a stored procedure not inside a query since there is no side effect detection mechanism currently implemented in Amos II.

The *callin* interface is similar to the call level interfaces for relational databases, such as ODBC, JDBC, etc. The callin interface is used when external programs written in

Java, C, or Lisp call Amos II. The *callout* interface makes it possible for AmosQL functions to call external functions written in Java, C, or Lisp.

A foreign function is defined in AmosQL using the following syntax for a foreign function implemented in Java:

```
create function <function name>(<argument declaration>)-> <result
declaration> as foreign ‘JAVA:<class file>/<method>;
```

<function name> is the name of the function, *<argument declaration>* is the declaration of the arguments to the foreign function, *<result declaration>* is the declaration of the result, *<class file>* is the name of the class where the method is implemented and *<method>* is the name of the entry point for the foreign function implemented in Java. If the class is in a package, it is specified by the package name and class file separated with a “.”.

The signature for a foreign function implemented in Java is:

```
public void foreignfunction(CallContext cxt, Tuple tpl) throws
AmosException;
```

Where *CallContext* is a data structure that manages the function call and *Tuple* is representing both arguments sent to the function and the results returned by it.

An optional *cost hint* can also be declared for a foreign function. The cost hint is an estimate of the cost to execute the function and is used by the query optimizer to choose the most efficient way to execute a query where a foreign function is involved [10].

3.5. XML

XML is a markup language for documents containing structured information. The XML specification is maintained by the W3C. The driving force behind XML and other similar technologies is the desire to exchange information in an open and nonproprietary manner. XML is a meta-language for describing markup languages and specifies neither semantics nor a tag set. It merely provides a facility to define tags and the structural relationships between them. The semantics of an XML document is defined by the processing application.

XML was created because at the time existing technology for describing document structure, *Standard Generalized Markup Language* (SGML) [31], was too hard to implement and too large for just presenting structured documents on the web. HTML [2] is a markup language that is designed to present information on the web. However, HTML does not have the ability to store data and metadata. SGML provides a common, but rather complex, format for defining and exchanging markups between systems that might not share the same markup language, XML which is a subset of SGML, takes the best features of SGML allowing it to marking up data in a standard, generalized way, but strips out the complexities that make SGML hard to implement. [32]

3.5.1. Well-formed XML documents

An XML document is considered being well-formed if it conforms to the XML syntax. An XML document is by definition well-formed. Hence, if a document is not well-formed it is not XML.
An XML document must conform to following syntax rules [33]:

- must begin with the XML declaration
- must have one unique root element
- all start tags must match end-tags
- XML tags are case sensitive

- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted

3.5.2. Valid XML documents

XML instance documents can be well-formed and still contain errors. For an XML instance document to be valid, it has to contain a proper *Document Type Declaration*[7] and conform to the constraints declared within. Most commonly, the constraints are expressed as a DTD definition or an XML Schema definition. The DTD definition language is a part of the XML recommendation and is inherited from SGML. DTD definition makes it possible to define the structure of an XML document. Making sure that an XML instance document is valid significantly improves the quality of document processing. Several tools for validating XML instance documents are available both as stand-alone programs and as programmatic APIs. An XML instance document that conforms to a particular XML Schema definition is said to be an XML instance document of that particular XML Schema definition.

3.5.3. XML namespaces

To make it possible to reuse previously defined markup vocabulary that has been previously defined, and eliminate the problems with name collisions for elements and attributes, W3C have introduced *XML namespace* [34]. The XML namespaces recommendation defines a way to distinguish between duplicate element type and attribute names. A namespace in XML is a collection of element type and attribute names identified by a unique URI. An element or attribute can uniquely be identified by a name that consists of two parts: the name of the namespace and the local name. The two-part name is referred to as a qualified name. [34]

An example of a well-formed, valid XML document is shown below in Figure 3. The document conforms to an XML Schema definition, *DCMDAddr.xsd*, which is a part of the XBench benchmark family.

```
<?xml version="1.0" encoding="UTF-8"?>
<addresses xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="DCMDAddr.xsd">
    <address id="1">
        <street_address>Department of Information
          Technology</street_address>
        <street_address>Lägerhyddsvägen 2</street_address>
        <name_of_city>Uppsala</name_of_city>
        <name_of_state>Uppland</name_of_state>
        <zip_code>751 05</zip_code>
        <country_id>89</country_id>
    </address>
    <address id="3">
        <street_address>Uppsala University School of
          Engineering</street_address>
        <street_address>Lägerhyddsvägen 1</street_address>
        <name_of_city>Uppsala</name_of_city>
        <name_of_state>Uppland</name_of_state>
        <zip_code>751 21</zip_code>
        <country_id>89</country_id>
    </address>
</addresses>
```

Figure 3: An example of a well-formed XML document.

[7] The Document Type Declaration must not be confused with the Document Type Definition. The former is used to identify and name the XML content, where as the latter is used to validate the metadata contained within [32].

3.6. XML Schema

The XML Schema is a set of recommendations from the W3C. The XML Schema provides means for defining the structure, content and semantics of XML instance documents. With XML Schema definitions, it is possible to model how the data in the XML instance document is to be represented and how data is related to each other i.e. parent/child, sibling relationship. The XML Schema definition language also makes it possible to define the datatypes of the data and is extensible because it is composed of XML-syntax. The predecessor DTD definition language, is composed of non-XML syntax, hence it is non-extensible which implies that it will constrain the evolution of XML. The XML Schema definition language has support for namespaces, which the predecessor has not. XML Schema definition language is a key component of Web Services specifications such as SOAP[8] and WSDL[9] [41], and is widely used to describe XML vocabularies precisely. The current recommendation, version 1.0, is from May 2001 and work on the next recommendation, version 1.1, is in progress. The XML Schema recommendation is rather extensive so the background information given here will only contain the most necessary parts for this thesis.

Figure 4 below shows the XML Schema definition *DCMDAddr.xsd*, a schema that is included from the XBench benchmark family.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="address">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="street_address" maxOccurs="2"
                    minOccurs="2"/>
                <xs:element ref="name_of_city"/>
                <xs:element ref="name_of_state"/>
                <xs:element ref="zip_code"/>
                <xs:element ref="country_id"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:long" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="addresses">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="address" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="country_id" type="xs:int"/>
    <xs:element name="name_of_city" type="xs:string"/>
    <xs:element name="name_of_state" type="xs:string"/>
    <xs:element name="street_address" type="xs:string"/>
    <xs:element name="zip_code" type="xs:string"/>
</xs:schema>
```

Figure 4: An XML Schema definition from the XBench benchmark family.

The XML Schema recommendation consists of three parts that are all publicly available via the Internet[10]:

[8] SOAP provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment, http://www.w3.org/2000/xp/Group/

[9]WSDL, Web Services Description Language, an XML language for describing Web services http://www.w3.org/2002/ws/desc/

[10] http://www.w3.org/XML/Schema

- *XML Schema Part 0: Primer* - a non-normative document that intends to provide a straightforwardly comprehensible description of the XML Schema definition language. [3]
- *XML Schema Part 1: Structures* – a normative document that specifies the XML Schema definition language, which offers facilities for describing the structure and constraining the contents of XML 1.0 documents. [4]
- *XML Schema Part 2: Datatypes* – a normative document that defines facilities for defining datatypes to be used in XML Schema definitions. [5]

3.6.1. Validation

XML Schema definitions are most commonly used to validate XML instance documents. Several available XML parsers and programs offer validation against XML Schema definitions as an option. A validating parser ensures that the XML instance document conforms to a specified XML Schema definition by controlling its structures and datatypes against the definitions in the XML Schema, see Figure 5 below. *Xerces2* is a fully compliant XML parser from the Apache[11] community of open-source software projects. It has implementations in C++ and Java. Another compliant XML parser is *MSXML* from Microsoft[12]. Other validation parsers and programs are listed on the web site of W3C[13].

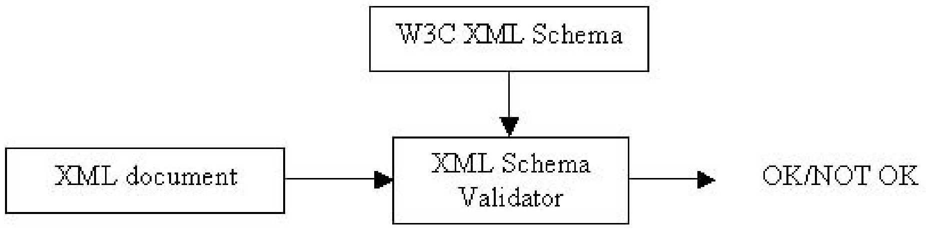

Figure 5: Validation of an XML instance document.

3.6.2. Structures

A XML Schema definition consists of different schema components divided into three groups: Primary-, Secondary- and Helper-components [4]. The primary components are necessary for the XML Schema definition language. Hence, the primary components are essential to this thesis. The primary component group consists of:

- Simple type definitions
- Complex type definitions
- Attribute declarations
- Element declarations

An XML Schema definition is always defined between the root element tag <*schema*> and <*/schema*>. Elements defined directly under the root is said to be global whereas when an element is defined as a sub-element is considered being local. The style of defining elements on a local level is often referred to as a *Russian doll design* [6]. Global definitions can be referenced directly in an element definition using the *type* attribute. Using such a design with global elements, give the XML Schema definition a modularity that the Russian doll design does not offer. All global definitions have to have names; however, local elements can be defined with or without names. Elements defined

[11] http://www.apache.org
[12] http://www.microsoft.com
[13] http://www.w3.org/XML/Schema

without names are called *anonymous* and the elements with a name *named*. All elements that are defined with the <*element*> element will be visible tags in the XML instance document. [4]

Simple type definitions

Simple types describe the contents of text nodes or attribute values and are independent of other nodes, thus independent of the XML markup. A simple type can only contain character data and no elements and cannot have any attributes. There are three means in the XML Schema definition language to define custom datatypes that use the built-in datatypes as a starting point: derivation by restriction, derivation by list and derivation by union [6]. When a datatype is derived by restriction, using available facets or regular expressions, it merely adds constraints and keeps the semantic and meaning of the original datatype it is derived from. A datatype that is derived by list has the semantic of a list and contains a list of values belonging to a datatype. The datatypes that are derived by union allows for having values belonging to different datatypes. Figure 6 defines a simple datatype named farenheitWaterTemp that is derived by restriction from the built-in datatype *xs:number*, it has two fractional digits and the value has to be in the interval
0.00 < value < 100.00.

```
<xs:simpleType name="farenheitWaterTemp">
  <xs:restriction base="xs:number">
    <xs:fractionDigits value="2"/>
    <xs:minExclusive value="0.00"/>
    <xs:maxExclusive value="100.00"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 6: A simpleType definition derived by restriction. [4]

Complex type definitions

A complex type defines constraints of the XML markup. A complex type can have different content models; complex when only sub-elements are expected, simple when only text nodes are expected, mixed when both text nodes and sub-elements can exist, or empty when only attributes are accepted [6]. Hence, the content model specifies which text nodes and sub-elements that an element can contain. There are two main ways to define a complex type: one for complex content models and one for simple content models. Simple content complex types are created by extending a simple type with attributes. There are two different ways to extend a complex type with a simple content model: derivation by extension and derivation by restriction. A derivation by extension merely adds an attribute to base type it is extending and cannot restrict the datatype of the text node nor the type of an attribute defined in its base type. However, a derivation by restrictions offers those facilities and can remove attributes that are not compulsory in the base type. Derivation of a complex type with a complex, mixed or empty content model is also possible through extension and restriction. Figure 7 shows a complexType definition with complex content model. [6]

```
<xs:complexType name="PurchaseOrderType">
  <xs:sequence>
    <xs:element name="shipTo" type="USAddress"/>
    <xs:element name="billTo" type="USAddress"/>
    <xs:element ref="comment" minOccurs="0"/>
    <xs:element name="items"  type="Items"/>
  </xs:sequence>
  <xs:attribute name="orderDate" type="xs:date"/>
</xs:complexType>
```

Figure 7: A complexType definition with a complex content model. [4]

Element declaration

An element declaration is an association of a name with either a simple or complex type definition. The association can be either global with the *ref* attribute or scoped to a containing complex type definition. Element declaration will appear as tags in the XML instance document. Figure 8 shows two different element declarations.

```
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>

<xs:element name="gift">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="birthday" type="xs:date"/>
      <xs:element ref="PurchaseOrder"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 8: XML representations of two different types of element declaration. [4]

Attribute declaration

Elements in an XML Schema definition can have attributes. This is indicated with the <*attribute*> element. An attribute can only be associated with a complex type, either explicitly or with a reference with the ref attribute. Figure 9 shows how an element address is defined as having an attribute id.

```
<xs:element name="address">
  <xs:complexType>
      <xs:sequence>
        <xs:element ref="street_address" maxOccurs="2" minOccurs="2"/>
        <xs:element ref="name_of_city"/>
        <xs:element ref="name_of_state"/>
        <xs:element ref="zip_code"/>
        <xs:element ref="country_id"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:long" use="required"/>
  </xs:complexType>
</xs:element>
```

Figure 9: Use of an attribute definition in a complex type definition.

Built-in datatypes

The XML Schema definition language defines a wide set of built-in simple datatypes. The datatypes are grouped into two groups: *Primitive datatypes* and *Derived datatypes*. The primitive datatypes are a smaller set of datatypes that have a specific meaning and semantic that cannot be derived from other datatypes. Derived datatypes are types derived from those primitive datatypes [6]. Currently the set of built-in datatypes in the XML Schema recommendation consists of 19 primitive and 25 derived datatypes. The derived datatypes are mostly derived by restriction, although some are derived by list, see Figure 10. None of the current built-in datatypes are derived by union.

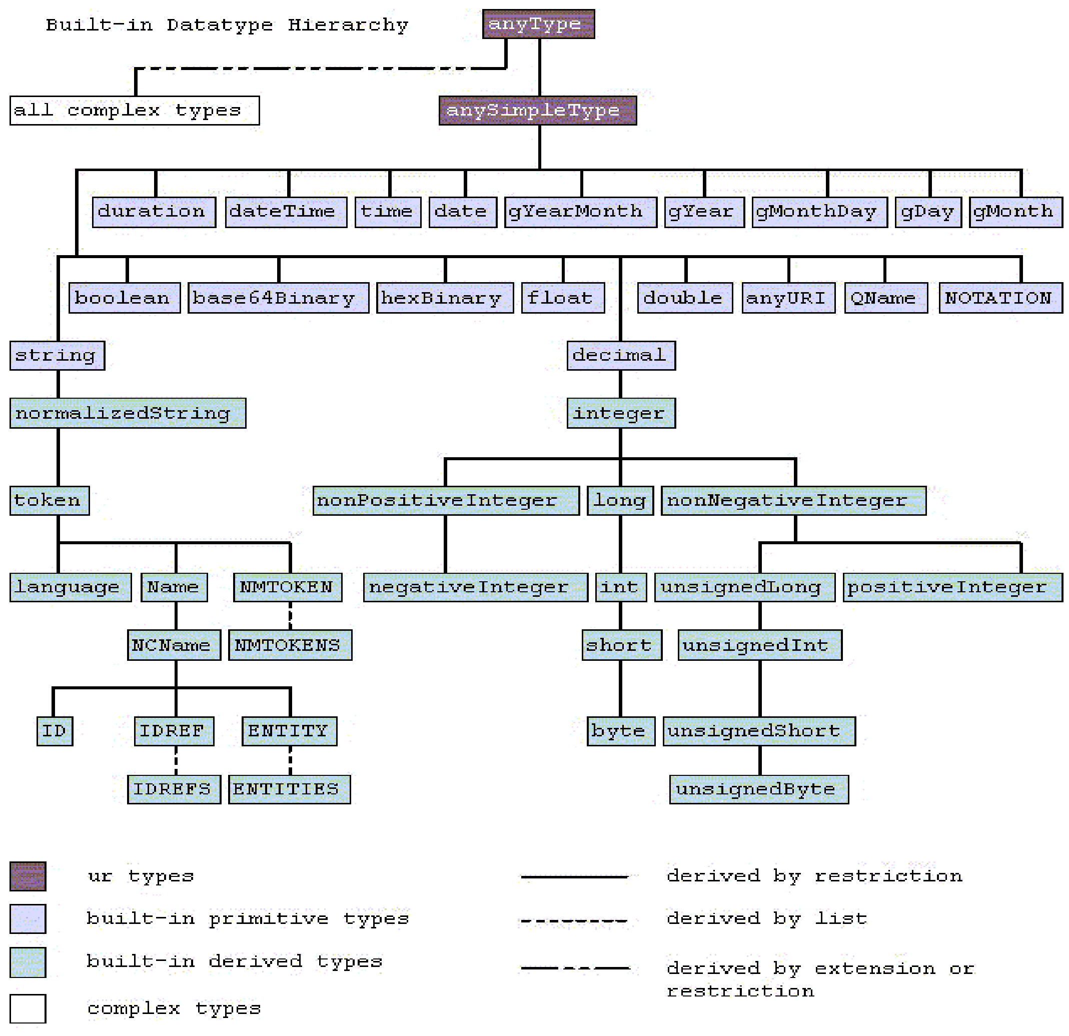

Figure 10: Built-in Datatype Hierarchy. [5]

The recommendation "XML Schema Part 2: Datatypes" defines the properties and behavior of each one of the built-in simple datatypes. It makes a distinction between the lexical space and the value space of the datatype. The lexical space defines the set of valid characters for a datatype used in the XML instance document. The value space is the set of values for a datatype. In some cases, the value space and the lexical space for a datatype are the same, and in sometimes they differ. For example, the value 3.1415 can be entered as 3.1415 or 31415E-4 for the built-in float datatype. [5]

Below follows a listing of the built-in simple datatypes with a short description. For the complete definition of the built-in simple datatypes, the reader is referred to the recommendation. [5] [6]

String datatypes

The string datatypes are derived from or have similar behavior as the simple built-in datatype *string*.

Primitive types:

- *string* – Its lexical space consist of tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646. Its value space is the set of finite-length sequences of characters from its lexical space. Derived type: *normalizedString*

- *QName* – Supports the use of XML namespace-prefixed names. It consists of a namespace part, anyURI, and a local part, NCName, separated with a colon ‘:’. The namespace part is optional and can be left out.
- anyURI – Represents a Uniform Resource Identifier, URI.
- *NOTATION* – Implements XML 1.0 second edition attribute *NOTATION*, can only be used in user-defined types.
- *hexBinary* – Used to string-encode binary content in an XML instance document. The value space is the set of finite-length sequences of bytes.
- *base64Binary* - Used to string-encode binary content in an XML instance document. The value space is the set of finite-length sequences of bytes.

Derived types:

- *normalizedString* – Derived from string, any occurrence of tab, linefeed or carriage return is replaced by space.
- *token* – Derived from normalizedString, trailing spaces are removed and continuous sequences of spaces are replaced with single spaces. Derived types: *language, Name, NMTOKEN.*
- *language* – Derived from token, accepts the standardized language codes in RFC 1766.
- *Name* – Derived from token, must start with a letter or the characters ‘:’ or ‘-‘. Derived type: *NCName*
- *NMTOKEN* – Derived from token, a single name token that consist of a set of allowed characters. Derived types: *NMTOKENS.*
- *NCName* – Derived from name, specifies a Name without a colon ‘:’. Derived types: *ID, IDREF, ENTITY.*
- *ID* – Derived from NCName, no duplicate of an ID can exist in an instance document. Hence, NCName can be used as a unique identifier in an instance document.
- *IDREF* – Derived from NCName, a reference to an ID, that must exist, in the same document. Derived type: *IDREFS.*
- *ENTITY*- Derived from NCName, must match an unparsed[14] entity in a DTD. Derived type: *ENTITIES.*

Numeric datatypes

Four primitive datatypes can be categorized as numeric datatypes.

Primitive types:

- *decimal* - Arbitrary long decimal number. Derived type: *integer*
- *float* - IEEE 32 bits floating-point.
- *double* - IEEE 64 bits floating-point.
- *boolean* - Valid literals are true, false, 1 and 0.

Derived types:

The derived numeric datatypes are presented in Figure 11 below with their minimum and maximum values. For derivates for each datatype see Figure 10.

[14] http://www.w3.org/TR/2000/WD-xml-2e-20000814#dt-unparsed

Type:	Minimum value:	Maximum value:
integer	-INF	INF
nonPositiveInteger	-INF	0
negativeInteger	-INF	-1
nonNegativeInteger	0	INF
positiveInteger	1	INF
long (64 bits)	-9223372036854775808	9223372036854775807
int (32 bits)	-2147483648	2147483647
short (16 bits)	-32768	32767
byte (8 bits)	-128	127
unsignedLong (64 bits)	0	18446744073709551615
unsignedInt (32 bits)	0	4294967295
unsignedShort (16 bits)	0	65535
unsignedByte (8 bits)	0	255

Figure 11: Derived numeric datatypes.

Date and time datatypes

The recommendation consists of a set of nine primitive datatypes for describing time [6]. The time datatypes relies on a subset of the standard ISO 8601, which is a solution from ISO for the confusion between different time and date formats around the world.

- *duration* – Represents a duration of time in the format PnYn MnDTnH nMnS where nY is the number of years, Mn is the number of Months and so on.
- *dateTime* – Defines a point in time in the format CCYY-MM-DDThh:mm:ss, where CC denotes century, YY year, MM month, DD date, hh hour, mm minute ,ss second. The letter T separates the date from the time part. The dateTime datatype also has an optional factional part for the seconds and a time zone.
- *time* – Represents a reoccurring point in time in the format hh:mm:ss with an optional time zone part.
- *date* – Represents a date in the format CCYY-MM-DD with an optional time zone part.
- *gYearMonth* – Represents a year and a month in the Gregorian calendar in the format CCYY-MM with an optional time zone part.
- *gYear* - Represents a year in the Gregorian calendar in the format CCYY with an optional time zone part.
- *gMonthDay* - Represents a month and a day in the Gregorian calendar in the format --MM-DD with an optional time zone part.
- *gDay* - Represents a day in the Gregorian calendar in the format ----DD with an optional time zone part.
- *gMonth* - Represents a month in the Gregorian calendar in the format --MM with an optional time zone part.

List types

There are currently three list datatypes in the XML Schema recommendation. Each one of them specifies a set of infinite, non-zero-length sequences of an IDREF, ENTITY or NMTOKEN. The three list datatypes are IDREFS, ENTITIES and NMTOKEN [6].

3.6.3. Namespaces in XML Schema

The XML Schema definition language offers support for namespaces to distinguish between different XML Schema definition vocabularies. Assigning attributes, elements, simple and complex types to a namespace is done by adding a prefix. The prefix is considered being a local shortcut for the URI, which is the real identifier for the namespace. If the prefix is left out, the elements do not belong to any namespace or they belong to the default namespace if such is defined. The default namespace does not apply to attributes. The definitions of the namespace prefixes are done as an attribute in the *<schema>* element. [6]
Examples:
Defining a default namespace:
```
<schema “xmlns=http://www.w3.org/2001/XMLSchema“....>...</schema>
```
Assigning a namespace to a prefix:
```
<schema “xmlns:udbl=http://user.it.uu.se/~udbl/“....>...</schema>
```
The targetNamespace attribute is used to define which namespace a schema describes:
```
<schema targetNamespace=“udbl=http://user.it.uu.se/~udbl/“....>
...</schema>
```

3.7. Benchmarks

There have been five different benchmarks proposed to test the efficiency of XML databases. They all have different approaches about how to measure efficiency. The different benchmarks can be classified into two groups: Application-level benchmarks and Micro benchmarks. The former focus on mimicking real world applications such as web applications whereas the latter concentrates on the basic query evaluation operations such as selections, joins and aggregations. The application-level benchmarks are valuable for testing and comparing how different XML databases system would perform against data and queries in a targeted XML application. Micro benchmarks invaluable engineering tools to measure the performance of individual operators and access methods [35].

3.7.1. XBench

XBench is a family of benchmarks from the University of Waterloo that recognizes that different applications require different benchmarks. It characterizes database applications along two dimensions: data characteristics and application characteristics. An application can be either data-centric or text-centric. Data-centric applications deal with data that might not originally be in XML, such as data for an e-commerce catalog or transactional data captured as XML. Text-centric applications handle actual text data natively encoded as XML instance documents such as dictionaries or book collections in a digital library. XBench generates text-centric and data-centric XML instance documents that conform to XML Schema definitions and DTD definitions. The XML Schema definitions, DTD definitions and workload queries specified in XQuery[15], are included in the benchmark that can be downloaded from the web [12].

3.7.2. XMach-1

XMach-1 generates XML data that models data from particular Internet applications. The data in XMach-1 is based on a web application that consists of text documents, schema-less data, and structured data. The data is generated with the help DTD definitions [36].

[15] http://www.w3.org/TR/xquery/

3.7.3. Xmark

The data in *XMark* is based on an Internet auction application that consists of fairly structured and data-oriented parts. It uses an XML data generator called *xmlgen* that generates documents according to a DTD definition [37].

3.7.4. XOO7

XOO7 is a benchmark for evaluating query-processing capabilities for XML management systems. It is an XML version of OO7, which is a benchmark for object-oriented database systems. The OO7 schema and instances are mapped into a DTD definition and eight queries translated into three different query languages [38].

3.7.5. The Michigan Benchmark

The Michigan benchmark is a micro benchmark that focuses on basic query evaluation operations such as selections, joins and aggregations. It primarily attempts to capture the rich variety of data structures and distributions possible in XML without mimicking any particular application. The benchmark specifies a single data set, which conforms to an XML Schema definition, against which carefully specified queries can be used to evaluate system performance for XML data with various characteristics [39].

Chapter 4

Realization

This chapter describes solutions to faced questions and problems when importing an XML Schema definition into an object-oriented database mediator system. Some solutions are based upon the background from the previous chapter.

4.1. Importing an XML Schema

The XML Schema recommendations that W3C provides split the definition language into two separate parts. The parts are the *XML Schema part 1: Structures recommendation* [4] and the *XML Schema part 2: Datatypes recommendation* [5]. Hence, the XML Schema definition language is about both structure and datatyping which in fact are relatively independent from each other. In addition, there is a big difference between the simple types, which deal with constraining content of the leaf nodes in an XML instance documents and complex types, which are about defining the structure of documents. A separation between translating structure and mapping datatypes in an XML Schema definition also seems appropriate to use in the tool architecture where it is possible. The tool needs to translate the structure of an XML Schema definition into an object-oriented database mediator database schema in a way that reflects the intended meaning of XML Schema definition. The declared datatypes therein also needs to map to corresponding datatypes in the mediator. It is vital to recognize that importing an XML Schema definition into an object-oriented database mediator system is only concerned with the data that the XML Schema definition represents and not the physical structure of the XML Schema definition document [21]. Hence, the tool needs to bind the XML instance document to a database schema in the mediator using schema definition statement used by the mediator.

A challenging topic is how the tool will describe the XML Schema definition language in the object-oriented mediator database data model. The first thing to do is to identify the components that govern the data model of an XML Schema definition, which consists of different components classified in different groups accordingly to *XML Schema part 1: Structures recommendation* [4]. The groups are Primary- Secondary- and the Helper-components. The primary components are necessary for the XML Schema definition language and hence, the tool focuses on translating the primary components group, which contains:

- simple- and complex type definitions.
- attribute- and element declarations.

The prototype tool generates translations accordingly to *translation rules* and these translations will extend the representation of the XML Schema definition in the database schema successively using related programming structures known to the mediator. In addition, the prototype tool performs the *mapping* of datatypes in the XML Schema definition to datatypes used by the database mediator. The mapping is as straightforward as possible but when no appropriate mapping is available, the tool needs other solutions.

Since an XML Schema definition is object-oriented in nature, a mediator system that shows the same behavior is preferable. The Amos II system is truly an object-oriented database mediator system that is publicly available. The Amos II system supports an advanced object-oriented query language called AmosQL and the prototype tool can use the programming structures of AmosQL to perform the importation of XML Schema definitions into the Amos II system.

4.2. Analyzing the XML Schema files in XBench

The XML Schema definitions included in the XBench benchmark consist of nine XML Schema definitions: *DCMDAddr.xsd*, *DCMDAuth.xsd*, *DCMDCoun.xsd*, *DCMDCust.xsd*, *DCMDItem.xsd*, *DCMDOrd.xsd*, *DCSD.xsd*, *TCMD.xsd* and *TCSD.xsd*. The majority of the XML Schema definition files define a document-centric (DCxx) XML structure and only two XML Schema definitions define a text-centric (TCxx) structure. The analysis is done by carefully reading the XML Schema definitions and looking for structures, models and datatypes that are used. The result will be used as a basis for the mapping of the datatypes and when writing the translation rules of the structures.

4.2.1. Namespace

None of the XML Schema definitions uses any other namespace than their default namespace, "http://www.w3.org/2001/XMLSchema".

4.2.2. Structures

The document-centric XML Schema definitions are written with modularity in mind; hence they do not use the Russian doll design. Elements are defined on global level and elements within *complex types* reference those with the '*ref*' attribute. The content models used for the complex types are *sequence*, *all*, *choice* and *mixed*. Only anonymous complex and simple types are used, i.e. declared within an element declaration [6]. The only derivation that is used is by restricting simple built-in types.

4.2.3. Datatypes

The analysis of the datatypes used in the XML Schema definitions shows that the majority are user-defined datatypes, i.e. complex or simple types or global elements [6]. The most common built-in simple type is the *string* datatype. The result of the analysis is presented below as a pie chart, see Figure 12, and a table with the actual number of occurrences of each datatype are shown in Figure 13.

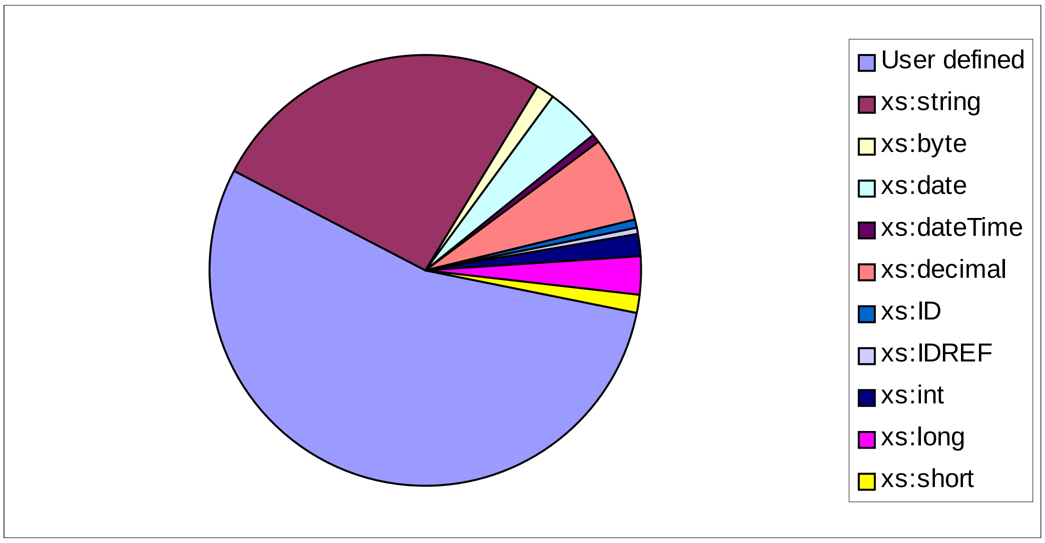

Figure 12: Datatypes used in the XBench benchmark's XML Schema definitions.

Datatype:	**Occurrence**:
User-defined	186
xs:string	91
xs:decimal	22
xs:date	14
xs:long	9
xs:int	6
xs:byte	5
xs:short	5
xs:dateTime	2
xs:ID	2
xs:IDREF	2

Figure 13: Datatype occurrences in numbers.

The major effort will be put into creating mappings for the most numerous datatypes. The mappings should be as near as possible to the defined system types in the Amos II system, see Figure 1. The reason for doing this is that it is the best way of preserving the good performance of the of the Amos II system. The user-defined types, which are the most numerous, are handled by the translation rules which are described below. The user-defined types cannot be mapped directly to literal system types in the Amos II system since they have no equivalent. Instead they are Types in the Amos II system with a behavior.

4.3. Translation rules

The following is a presentation over the translation rules that the tool uses to import an XML Schema definition into the object-oriented mediator system Amos II. The existing translation rules are given and examples in AmosQL syntax show how the tool would translate an XML Schema definition statement using a relevant rule and Amos II programming structure into the Amos II mediator system. Most of the example statements are contained in the XML Schema definition file DCSD.xsd, which is from the XBench benchmark suite. Figure 14 below shows a partial extract from that file.

```
<?XML version="1.0" encoding="UTF-8"?>
<xs:schema XMLns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="FAX_number" type="xs:string"/>
  ...
  <xs:element name="author">
    <xs:complexType>
      <xs:all>
        <xs:element name="name">
          <xs:complexType>
            <xs:all>
              <xs:element ref="first_name"/>
              <xs:element ref="middle_name"/>
              <xs:element ref="last_name"/>
            </xs:all>
          </xs:complexType>
        </xs:element>
        <xs:element ref="date_of_birth"/>
        <xs:element ref="biography"/>
        <xs:element name="contact_information">
          <xs:complexType>
            <xs:all>
              <xs:element name="mailing_address">
                <xs:complexType>
                  <xs:all>
                    <xs:element ref="street_information"/>
                    <xs:element ref="name_of_city"/>
                    <xs:element ref="name_of_state"/>
                    <xs:element ref="zip_code"/>
                    <xs:element name="name_of_country" type="xs:string"/>
                  </xs:all>
                </xs:complexType>
              </xs:element>
              <xs:element ref="phone_number"/>
            <xs:element ref="email_address"/>
          </xs:all>
        </xs:complexType>
      </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>
...
</xs:schema>
```

Figure 14: Part of DSCD.xsd from the XBench benchmark suite.

The first rule creates a user-defined type [9] in the database schema for a global element declaration in the XML Schema definition:

Rule 1. For every <element> element information item that has the <schema> element formation item as its parent a new type is created.

Assume, from Figure 14, the following element declaration:

```
<xs:element name="FAX_number" type="xs:string"/>
```

This element declaration and several other element declarations have the <schema> component as parent and therefore, are global element declarations [3]. All global element declarations can be root in an XML instance document and thus must be a type in the Amos II system. Rule 1 would instruct Amos II to extend the database schema, using AmosQL [8] [9], with a new type named *FAX_number*. The new type is set to be a subtype of the user-defined type *XML*.

```
create type FAX_number under XML;
```

The extent of the type *XML* in the database schema will be all created types completed by tool. This can be useful if sub-types of XML must be identified.

The second rule creates a property function [8], instead of a type, based on a type definition of an element and adds it to the database schema:

Rule 2. For every global element definition, E with a given type [attribute] creates a property function with signature E (E) ->E.

Consider the same element declaration as above:

```
<xs:element name="FAX_number" type="xs:string"/>
```

Rule 2, instructs the Amos II system to dynamically create a property function named *FAX_number* with a *FAX_number* object as argument and returning the built-in type *Charstring*. In AmosQL this can be expressed as:

```
create function FAX_number(FAX_number) -> charstring as stored;
```

For built-in type definitions in the XML Schema definition language, i.e. *string*, *integer*, *duration* etc, a mapping is performed. How the mapping is performed is described under section 4.4.

The third rule creates a user-defined type in the database schema for each global element declaration in the schema:

Rule 3. For every <element> element information item that is defined as a <complexType> and has <complexType> as ancestor, a new type is created.

Assume, from Figure 14the following element declaration:

```
<xs:element name="name">
```

When rule 3 is applied a new type, *name* will extend the database schema:

```
create type name under XML;
```

The fourth rule would then create containment functions based on the element type definitions and add this to the database schema:

Rule 4. For every element declaration E, defined as a complex type that is contained within another complex type definition E1 a containment function is created with signature E (E1) -> E. It returns the sub-element E of a given parent element E1.

When rule 4 is applied on the element declaration Amos II dynamically adds the function:

```
create function name(author) -> name as stored;
```

When the tool creates the Amos II statements, the declaration of the complex type definition will be used for the parent element. The reason for this is, that definitions themselves will not be completely visible in the instance document only the declarations. The visible parts of a definition are the order of its sub-elements.

When a schema contains a local declaration defined as a built-in type a property function will be created. The rule below explains what happens:

Rule 5. Every element declaration E that represents a built-in type and also is part of a complex type definition E1 creates a property function with signature E (E1) -> E on the parent element E1. It returns the value of E contained in a given object E1.

The tool would use rule five when the following declaration from Figure 14 is encountered.

```
<xs:element name="name_of_country" type="xs:string"/>
```

The result from applying rule 5 to this element declaration is an Amos II property function on the parent element.

```
create function name_of_country(mailing_address) -> charstring as
stored;
```

Figure 14 contains many examples of declaring local elements using the ref attribute. A ref attribute can be used to refer to other global element declarations. This means that the referred elements must be created as types at some point by the tool, since they are global. When the tool encounters the local elements declarations using the ref attribute, the following rule is used:

Rule 6. For every element declaration E, declared with the ref [attribute], which is contained within another complex type definition E1 a containment function is created with signature E (E1) -> E. It returns the sub-element E of a given parent element E1.

The following statement is taken from Figure 14:

```
<xs:element ref="first_name"/>
```

Accordingly to rule 6 the following Amos II containment function is created:

```
create function first_name(name) -> first_name as stored;
```

The tool does not create a type for *first_name* here, instead it creates a function. The type is either already created, if the declaration of first_name occurred before the ref declaration, or will be created later when the tool encounters the global element declaration for *first_name*.

```
<?XML version="1.0" encoding="UTF-8"?>
<xs:schema XMLns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  ...
  <xs:element name="height">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:decimal">
          <xs:attribute name="unit" type="xs:string" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>
```

Figure 15: Another part of DCSD.xsd from the XBench benchmark suite.

Components in the XML Schema definition that describe elements with attributes are always complex types, which contain an attribute declaration [6]. The attribute itself consists of a declaration containing its definition, which is a simple type. If an element declares an attribute, the tool will use rule 7 below.

Rule 7. Every defined attribute declaration A that is part of a complex type definition E creates an attribute function. The name of the function is also named A and its signature is A(E) -> A.

Consider the following statement taken from Figure 15:
`<xsd:attribute name="unit" type="xs:string" use="required"/>`

The tool will create the attribute function named *unit* and the database schema is extended by Amos II with:
`create function unit(height) -> charstring as stored;`

This represents values of the attribute *unit* for objects of type *height*.

Figure 15 also shows an example of the inheritance mechanism in the XML Schema definition language using the extension component. Amos II also has this mechanism but does not allow extension of primitive types into new user defined types. This would have been useful in this case. For cases where elements are declared as extensions of built-in types in an XML Schema definition, the tool will translate the extension component as a definition of *height* as being of type *decimal*. Hence, the tool creates a property function (Rule 2) in Amos II on the object *height*:
`create function height(height) -> real as stored;`

```
<xs:schema>
  ...
  <xs:element name="student" type="student"/>
  ...
  <xs:complexType name="student">
    <xs:complexContent>
      <xs:extension base="person">
        ...
  </xs:complexType>
  ...
  <xs:complexType name="person">
    <xs:sequence>
        ...
    </xs:sequence>
  </xs:complexType>
  ...
</xs:schema>
```

Figure 16: An example using the extension component.

If the extension is defined as the extension of a user-defined type on the other hand, a translation to Amos II is possible. Rule 8 shows this as:

Rule 8. For every defined extension E1, defined as a complex type, which has a complex type E as ancestor an extension of E with E1 is created. It has signature E under E1.

The element declaration *student*, from Figure 16 above, is an example of an extension of a complex type *person*. *Student* is defined as the extension of the complex type *person*. In such a case the type *person* must also be defined as a complex type somewhere in the XML Schema definition. The tool will create the following Amos II type:

```
create type student under person;
```

It is most likely that at some stage, the tool will create a set of functions for the super type *person*. In Figure 16 above, the tool creates the functions for *person* after the creation of *student* because the definition of *person* follows the definition of *student*. The other way around, i.e. the XML Schema definition defines *person* before *student* is also a possible. Because of the inheritance mechanisms in the Amos II system, the inherited functions from the super type *person* for the sub type *student* will be correct. The prototype tool has to make sure though, that the mediator system creates the super type before it creates a sub type for this to work.

4.4. Mapping XML Schema built-in datatypes to Amos II datatypes.

The XML Schema definitions included in the XBench benchmark only use a subset of the built-in datatypes in the XML Schema definition language. Some of the built-in datatypes in the XML Schema definition language have no equivalent in Amos II. For example, the W3C predefined *integer* can be arbitrary long and the *Integer* datatype in Amos II is 32 bits wide. This can cause a problem when the XML instance document contains data of type *integer* and is larger than what can fit in a 32 bits wide *Integer* in Amos II. However, the approach taken in this thesis is to keep it simple and efficient without having to extend the Amos II system with new literals. The mappings will be done to match the literals already defined in the Amos II system. Hence, some mappings will result in loss of precision.

What is important when mapping the datatypes is that the value spaces of the datatypes are the same, the lexical space, i.e. how the value is represented in the XML instance document, is of minor importance. Hence, this thesis will only consider the value space when designing the mappings between XML Schema definition datatypes and Amos II datatypes.

Ten built-in datatypes from the XBench XML Schema definition files need to be mapped to the Amos II system: *string*, *decimal*, *date*, *long*, *int*, *byte*, *short*, *dateTime*, *ID*, *IDREF*, see Figure 13 above.

4.4.1. Mapping string datatypes

The string datatypes in the XML Schema definition language are derived from or have similar behavior as the built-in simple type *string*, which is able to store a finite-length sequence of 16-bits Unicode characters. The *string* datatype in Amos II is the datatype *Charstring*, which is implemented as a sequence of bytes.

The primitive built-in *string* datatypes will be mapped to the Amos II *Charstring* datatype. This mapping will only cause problems when the *string* datatype consist characters outside the 8-bits ASCII table. For the rest of the primitive *string* datatypes, *QName*, *anyURI*, *NOTATION*, *hexBinary* and *base64Binary*, the mapping will not cause any problems.

The datatypes that are derived by restricting the *string* primitive will also be mapped to the Amos II *Charstring* datatype with the exception of *IDREF* and *ID*. The *IDREF* datatype is a reference in the XML instance document to an element containing an attribute of the type *ID*. It is not known in the XML Schema definition which element it will be referencing to until the XML instance document is created. It can point to any element that has an attribute of type *ID*. Hence, since the element is unknown, it has to be mapped to the created supertype *XML*, which all user defined types from the XML Schema definition is created under. If the *ID* datatype would be mapped to a *Charstring*, it would not be possible to later, when reading the XML instance document, map the *IDREF* to the *ID* since the information about which elements that contains an attribute of type *ID* is lost. Therefore, a user defined datatype *ID* is created that the *ID* datatype can be mapped to.

4.4.2. Mapping numeric datatypes

Floating-point datatypes

The numeric primitives in the XML Schema definition language are the *decimal* datatype, which can hold an arbitrary long decimal number, and the single- and double precision floating-point datatypes *float* and *double*. All other built-in numeric datatypes are derived by restricting the *decimal* datatype. The datatype that can hold a floating-point value in Amos II is datatype *Real*, which is a double precision floating point datatype. Therefore, the *decimal*, *float* and *double* datatypes will be mapped to the *Real* datatype. However, mapping the primitive *decimal* type to the *Real* datatype will result in a precision loss.

Integer datatypes

The *integer* datatype in the XML Schema definition language is derived from the *decimal* datatype by restricting it to have no fractional part. Hence, it can hold an arbitrary long integer number. The Amos II *Integer* datatype is a signed 32-bits wide datatype and will obviously not be able to hold the value of an arbitrary long integer. Therefore, to be able to store numbers larger than what the *Integer* datatype in Amos II can, the built-in *integer* datatype will be mapped to the *Real* datatype in Amos II. The integer datatypes *nonPositiveInteger*, *negativeInteger*, *nonNegativeInteger* and *positiveInteger* have infinity as either their upper or lower limit. They are all arbitrary long in one direction or another, and can therefore not be directly mapped to the *Integer* datatype. They will therefore be as the decimal and integer datatype, mapped to the *Real* datatype. The mapping to the *Real* datatype will of course not allow arbitrary long integer numbers to be stored. However, it can store a wider range of numbers than the *Intege*r datatype can.

Amos II does not have any literal that can store a 64-bits long integer or a 32-bits unsigned integer. Therefore, the datatypes *long*, *unsignedLong* and *unsignedInt* will to be mapped to the *Real* datatype. Some precision will be lost for large numbers.

The datatypes *int*, *short*, *byte*, *unsignedShort* and *unsignedByte* are defined as 32-, 16- and 8-bits integer numbers respectively. The *Integer* datatype in Amos II is 32-bits wide. Therefore, the datatypes integer *int*, *short* and *byte* will be mapped to the *Integer* datatype without any loss of precision.

Boolean datatype

The value spaces for the XML Schema definition language datatype *boolean* and *Boolean* in Amos II are identical {true, false}. Therefore, the built-in *boolean* datatype will be mapped to the *Boolean* datatype in the Amos II system.

4.4.3. Date and time datatypes

The only built-in *date* or *time* datatype used in the XBench benchmark is *dateTime*, which is a composition of the *date* and *time* built-in datatypes. Amos II has three date and time (temporal) datatype: *Date*, *Time* and *Timeval*. *Timeval* is able to store date and time.

Therefore, the *dateTime* datatype will be mapped to the *Timeval* datatype in Amos II. Unfortunately, the *Timeval* datatype is unable to store the optional fractional part of the seconds or information about time zones. The built-in datatypes *date* and *time* have the corresponding datatypes *Date* and *Time* in Amos II. The difference though is that none of the Amos II temporal types can store information about time zone. However, the *time* and *date* built-in datatypes will be mapped to the Amos II *Time* and *Date* datatypes.

The remainder of the built-in *time* and *date* datatypes will be mapped to *Charstring*, since no equivalent literal exist in Amos II an they do not occur in the XML Schema definitions from the XBench benchmark.

4.4.4. The resulting mapping

Figure 17 below shows the resulting mappings between the built-in simple types in the XML Schema definition language to Amos II literals.

W3C datatype	AMOS II datatype	W3C datatype	AMOS II datatype
anyURI	Charstring	integer	Real
Base64Binary	Charstring	language	charstring
boolean	Boolean	long	Integer
Byte	Integer	Name	charstring
Date	Date	NCName	charstring
dateTime	Charstring	negativeInteger	Real
decimal	Real	NMTOKEN	charstring
double	Real	NMTOKENS	charstring
duration	Charstring	nonNegativeInteger	Real
ENTITIES	Charstring	nonPositiveInteger	Real
ENTITY	Charstring	NormalizedString	charstring
Float	Real	NOTATION	charstring
gDay	Charstring	PositiveInteger	Real
gMonth	Charstring	QName	charstring
gMonthDay	Charstring	Short	Integer
gYear	Charstring	String	charstring
gYearMonth	Charstring	Time	Time
hexBinary	Charstring	Token	charstring
ID	XS_ID	UnsignedByte	Integer
IDREF	XML	UnsignedInt	Integer
IDREFS	XML	UnsignedLong	Integer
Int	Integer	UnsignedShort	Integer

Figure 17: Mappings between XML Schema definition language datatypes and Amos II datatypes.

4.5. Architecture

The general architecture of the prototype tool program module, shown in Figure 18 below, contains four separate modules specialized at doing some part of the XML Schema definition importation into Amos II. Separation of the architecture follows the separation of the XML Schema specification [4] [5]. Thus, the different modules are able to perform one particular part of the translation. The modules are AmosXSD, XSDTranslator, AmosResolver and AmosTypeMapper. The translator module also use external modules to parse XML Schema definition files that the Amos II system whish to import. The AmosXSD module is the entry point to the importer tool and it will hold application state for the tool. The translator module in turn, uses an external parser to read an XML Schema definition file, identify the contained structures, and decide what parts are represented as types or functions in the Amos II system using the translation rules. The translator then uses the resolver, which creates actual objects representing the identified types and functions in Amos II using the AmosTypeMapper to perform mappings of datatypes for the created functions. The resolver is also responsible for ordering the created objects in the correct order so that no problems occur when creating the database schema. The result of an importation will be a sorted list of objects that expressed in string representation is a list of AmosQL statements. The AmosXSD module then executes the AmosQL statements against the Amos II system, which will result in the creation of an imported database schema.

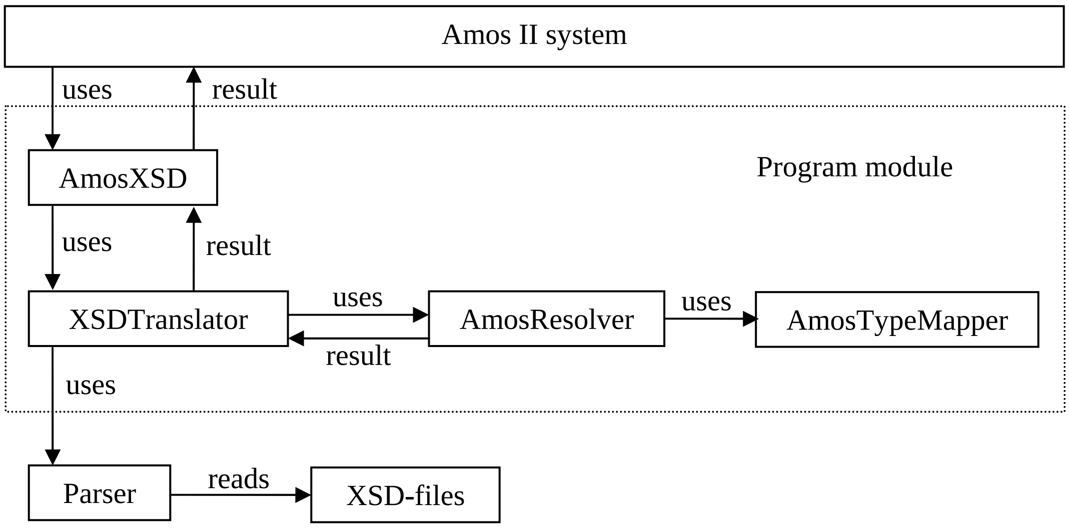

Figure 18: The tool's general design. The picture shows the different parts contained in the program module.

4.6. Implementation

In order to study methods for importing an XML Schema definition into the Amos II system an XML Schema definition importer prototype tool was implemented using the Java 2 Platform, Standard Edition (*J2SE*)[16]. The decision to implement the tool in java is based upon the ability of the Amos II system to define foreign functions written in java and the strong support the java platform has for XML with JAXP [14]. Two components, included in this API are, the org.w3c.dom package, which is an interface to DOM [16], and the org.xml.sax package, which provides interfaces for SAX [17]. The tool needs to be able to process XML in some way and the javax.xml.parsers package already provides a set of classes for processing XML documents using parsers. Two different types of pluggable parsers are available, a SAX parser and a DOM parser. Hence, no other means for processing XML documents are needed if the tool would use either of these.

The already available XML parsers were a crucial argument for using the java technology since the tool must be able to process XML in order to create database schemas in Amos II from a given XML Schema definition data source. Using the provided Java APIs with the tool also avoids the need to create a new program for the tool that process an XML file. The idea is to make the tool in a way that it can create either a database schema as a file that the Amos II system reads at a later stage or a database schema directly in the Amos II system, using translation rules and mapping of datatypes by using an XML parser.

[16] http://java.sun.com/j2se/

4.7. User interaction

Given an XML Schema definition, the tool automatically translates the XML Schema definition into an Amos II database schema using the Amos II data model. An XML Schema definition is given as an argument to a defined Amos II resolvent function, called *importXSD*, for the tool from the Amos II system. No other user interaction is required for the tool to produce a translation. Figure 19 is an example of using the tool.

```
D:\Program\AmosII\bin>javaamos

D:\Program\AmosII\bin>java JavaAMOS
JavaAMOS 1> create function importXSD(charstring)->charstring as
foreign "JAVA:xsd.AmosXSD/importXSD";
#[OID 729 "CHARSTRING.IMPORTXSD->CHARSTRING"]
JavaAMOS 2> importXSD("DCSD.xsd");
Importing W3C XML Schema DCSD.xsd...Expanding image to 4038137
Image moved in MAKEFN-INDEX
done!
0.161 s
JavaAMOS 3>
```

Figure 19: An example of running the tool.

Furthermore, the tool can selectively be executed as a stand-alone application in which case it produces a file containing the imported database schema rather then directly importing it into the Amos II system. The Amos II system can then read the file at a later stage.

Chapter 5

The implemented tool

The primary result of this thesis is the XML-enabled tool, which the Amos II system can use to translate an XML Schema definition into a database schema, expressed in the Amos II data model. Secondary results are the translation rules and the mapping of datatypes that the tool uses to import an XML Schema definition into the Amos II system. The XBench benchmark suite provides a set of XML Schema definitions that are used to test the tool and perform some experiments. This chapter describes the tool and the results of some of the more interesting experiments.

5.1. The Amos II XML Schema import tool (AXSI)

The *Amos II XML Schema import tool* (AXSI) is the result of the realization. The AXSI implements the translation rules and the mapping mechanism of the two separate data models found in XML Schema definition and the Amos II system.

5.2. Parser choice

JAXP in the J2SE platform provides support for processing XML files with parsers either by using the SAX or DOM APIs. AXSI can use the included parsers to read an XML Schema definition to avoid the development of a new program module that process the XML Schema definition. [14]

For the Amos II system, the interesting parts of an XML Schema definition are the constraints the definition represents for an XML instance document. In order to get hold of the constraints the AXSI will be XML-enabled with an XML parser to extract the information from an XML Schema definition. The tool implements the SAX APIs from the org.xml.sax package for this purpose. The SAX APIs let the AXSI register a SAX parser and read an XML Schema definition using callback methods. The sax parser is small and fast and will read an XML Schema definition from the beginning of the document until the end and notify the AXSI of element-by-element events such as the beginning of an element when the “<” symbol is encountered or end of an element when the “/>” symbol is encountered. [13]

The implementation of AXSI becomes somewhat more complex as a result of using SAX because it needs additional data structures in order to preserve order and relationship between elements. For example, the AXSI preserves the parent child relationship between elements with a stack implementation in the translator module.

The Apache Software Foundation has created two available parsers that support SAX, Crimson[17] and Xerces2[18]. Crimson is actually bundled with the J2SE 1.4 and later, to provide JAXP support. It is located in rt.jar, which is part of the *Java Runtime Environment* (JRE). This means that the AXSI will work with Crimson for anyone that uses J2SE 1.4 or later, without altering any implementation details for AXSI or the JRE. Xerces2 on the other hand, requires the *Endorsed Standards Override Mechanism*[19] in order for AXSI to work properly.

To use Crimson with the AXSI is a better choice than using Xerces2. Crimson is a straightforward implementation of an XML parser with a small footprint: approximately

[17] http://xml.apache.org/crimson/index.html
[18] http://xml.apache.org/xerces2-j/index.html
[19] http://java.sun.com/j2se/1.4.1/docs/guide/standards/

200KB (jar file size) while Xerces2 is more advanced and includes many additional features like XML Schema support to validate XML instance documents. Xerces2 also comes with support for WML and HTML DOMs, which increase the size of the jar file, to around 2.5MB [40]. The conclusion is to use the Crimson implementation with the AXSI cause of Crimson's availability in the JRE and its small footprint. Performance wise, there is hardly any difference between the two SAX parsers when used with the AXSI to import XML Schema definitions into the Amos II system.

5.3. Results from running the tool

To verify that the AXSI module creates correct translations of XML Schema definitions into Amos II database schemas the tool made several translations. The XML Schema definitions come from the XBench benchmark suite, and they provide a "white box" testing ground since their structure is known and an expected translation can be performed by hand in advance prior to letting the AXSI module create a translation. A comparison between the expected translation and the actual translation made by the AXSI module is then used to see if the AXSI's translation is correct. A change in the AXSI module was needed if the test failed.

Figure 20 shows one of the included XML Schema definitions, called *DCMDAddr.xsd*, which is used to verify the correctness of the output of AXSI.

```
<?XML version="1.0" encoding="UTF-8"?>
  <xs:schema XMLns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="address">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="street_address" minOccurs="2" maxOccurs="2"/>
          <xs:element ref="name_of_city"/>
          <xs:element ref="name_of_state"/>
          <xs:element ref="zip_code"/>
          <xs:element ref="country_id"/>
        </xs:sequence>
      <xs:attribute name="id" type="xs:long" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="addresses">
      <xs:complexType>
        <xs:sequence>
          <xs:element ref="address" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="country_id" type="xs:int"/>
    <xs:element name="name_of_city" type="xs:string"/>
    <xs:element name="name_of_state" type="xs:string"/>
    <xs:element name="street_address" type="xs:string"/>
    <xs:element name="zip_code" type="xs:string"/>
  </xs:schema>
```

Figure 20: DCMDAddr.xsd is an XML Schema definition from the XBench benchmark suite.

This XML Schema definition constraints an XML instance document for instance *Addresses.xml*, which is a made up XML instance document. Figure 21 shows the file Addresses.xml below.

```
<?XML version="1.0" encoding="UTF-8"?>
  <addresses XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="D:\Program\XBench\xbench\schemas\DCMDAdd
r.xsd">
    <address id="1">
```

```
      <street_address>Department of Information
        Technology</street_address>
      <street_address>Lägerhyddsvägen 2</street_address>
      <name_of_city>Uppsala</name_of_city>
      <name_of_state>Uppland</name_of_state>
      <zip_code>751 05</zip_code>
      <country_id>89</country_id>
    </address>
    <address id="3">
      <street_address>Uppsala University School of
        Engineering</street_address>
      <street_address>Lägerhyddsvägen 1</street_address>
      <name_of_city>Uppsala</name_of_city>
      <name_of_state>Uppland</name_of_state>
      <zip_code>751 21</zip_code>
      <country_id>89</country_id>
    </address>
  </addresses>
```

Figure 21: Addresses.xml is an example XML instance document that is valid against DCMDAddr.xsd.

The AXSI module creates the following Amos II database schema, shown in Figure 22 below, when the Amos II system calls the resolvent function *importXSD* with a URI to the DCMDAddr.xsd as argument. When the translation is finished, the AXSI module emits the statements back to the Amos II system and the XML Schema definition importation is complete.

```
create type XML;
create type XS_address under XML;
create type XS_addresses under XML;
create type XS_country_id under XML;
create type XS_name_of_city under XML;
create type XS_name_of_state under XML;
create type XS_street_address under XML;
create type XS_zip_code under XML;
create function XS_street_address(XS_address nonkey)->XS_street_address
as stored;
create function XS_name_of_city(XS_address)->XS_name_of_city as stored;
create function XS_name_of_state(XS_address)->XS_name_of_state as
stored;
create function XS_zip_code(XS_address)->XS_zip_code as stored;
create function XS_country_id(XS_address)->XS_country_id as stored;
create function XS_id(XS_address)->integer as stored;
create function XS_address(XS_addresses nonkey)->XS_address as stored;
create function XS_country_id(XS_country_id)->integer as stored;
create function XS_name_of_city(XS_name_of_city)->charstring as stored;
create function XS_name_of_state(XS_name_of_state)->charstring as
stored;
create function XS_street_address(XS_street_address)->charstring as
stored;
create function XS_zip_code(XS_zip_code)->charstring as stored;
```

Figure 22: A translation of the DCMDAddr.xsd XML Schema definition into an Amos II database schema.

The AXSI module always creates the user-defined type XML first because all other user-defined types will be subtypes of the type XML. Hence, the extent of the type XML will be all created types created afterwards. In addition, the AXSI module will add the prefix "XS_" throughout the translation to avoid naming conflicts with other pre-defined functions in the Amos II system. The AXSI module creates this database schema using the previously described translation rules and datatype mappings between the XML Schema definition language and the Amos II system. The AXSI module simply handles cardinality constraints in the XML Schema definition as either a one-to-one relationship or no cardinality constraint for the relationship by adding the *nonkey* to a function definition in the database schema [9]. Figure 22 above, shows this in the declaration of the first function declaration called XS_street_address. The AXSI module adds the nonkey since the DCMDAddr.xsd requires more than one occurrence of street_address.

5.4. Limitations

This section describes the current limitations of the AXSI implementation.

5.4.1. XML Schema mixed content model

The current implementation of the AXSI module does not support XML Schema definition mixed content models for XML instance documents. Figure 23 illustrates an arbitrary example of using a mixed content model for a tag <A> found in an XML instance document; in this example, this means that tag <A> contains a mixture of both text and child-elements tag <c> and tag <b>. This content model appears more frequently in text centric XML documents than in data centric XML documents where it is more common to use a content model containing either text or child-elements but not both at the same time.

```
<A>
  This text <c>cc</c> makes
  <b>bbbb</b> no sense
  <c>cccc</c> except as
  <b>bb</b> an example
</A>
```

Figure 23: An arbitrary example of a tag A that uses a mixed content model.

Currently the AXSI module will translate the mixed content of tag <A> as a type A having the functions XS_c and XS_b returning c and b instances and the text that otherwise occurs in tag <A> is disregarded.

A solution to the mixed content problem is to let AXSI use a Collection of unknown size, for instance the system datatype "Vector", in Amos II to wrap the elements contained in tag <A>. The Amos II vector datatype is able to preserve the order among contained elements as well. R. Bourrret discusses the problem and solution in detail in [20].

5.4.2. XML Schema model group compositors

The XML Schema definition language's model group component consists of one or more recursive compositors, which is either one of *all*, *choice* or *sequence* [4]. The AXSI module will translate every model group to contain all defined particles within a model group as types or functions with no specified order. Thus, there is a loss of semantics as the translation omits sequences or choices constraints.

5.4.3. Loss of schema specific details

The AXSI never translates the <schema> component and any children to the <schema> component containing schema specific details and consequently never imports it into the Amos II system. Information items regarding an imported XML Schema definition document type, such as *targetNamespace*, *version*, *notation definitions*, *annotation definition* and others is thus lost. The reason for disregarding this is that this is documents specific details and this is not of interest to the Amos II system [21]. AXSI identifies and imports the content of the <schema> component regarding primary components i.e. types-, elements-, and attribute definitions. If the XML Schema definition details were interesting however, the AXSI needs to define a new "schema" type in the Amos II system, define more translation rules for the information items and mappings of datatypes and add property functions for the "schema" type to store the XML Schema definition details in.

5.5. Alternative implementation

The AXSI module uses the SAX API in order to parse an XML Schema definition and translate the contained structures using translation rules. This is not the only solution that works to solve the problem of importing an XML Schema definition into the Amos II system, however. Alternative implementations from JAXP are to use perhaps the DOM API included in org.w3c.dom, or XSLT contained in the javax.xml.transform package or some other solution, for example using an XML Schema definition compiler that performs a direct data binding between an XML Schema definition and the Amos II system much in the same way JAXB works [15]. This thesis does not discuss the compiler approach, however since the AXSI has not been tested with this implementation. Instead, this is a short description for the pros and cons of using DOM [16] or XSLT [18] with the AXSI instead of using SAX. The AXSI module used these technologies at some point during development of the prototype; the final solution however, was to go with the SAX API.

5.5.1. Using the DOM API

The DOM APIs can be used to build an in memory object representation of an XML document. Since an XML Schema definition is a well-formed XML document, a DOM parser can create a DOM representation of the XML Schema definition. The representation is a tree data structure containing nodes that represent the entire XML Schema definition and once a parser builds a DOM out of the XML Schema definition, the tree allows random accesses to particular pieces of data with `get`, `set`, and `create` methods, like any other tree data structure.

Using the DOM API to create an in memory DOM representing an entire XML Schema definition, would allow the AXSI to navigate its structure and add, modify and delete elements, attributes and content interactively to produce a DOM which suite the Amos II system data model. For instance, the structure between elements like parent and child or siblings relations, are easily identified by other program modules in the DOM and this can be very helpful when extracting information from the result that translate to functions and types in the Amos II system.

However, the physical structure of an XML Schema definition document is not of particular interest to the AXSI, the data that the document constraints are important [21]. Below is a summary of the pros and cons of using DOM with the AXSI:

The pros of using the DOM APIs with the AXSI
DOM allows AXSI to create documents, navigate their structure and add, modify and delete elements, attributes and content.
DOM allows documents to be interactively modified.

The cons of using the DOM APIs with the AXSI
The DOM is an in memory model of a parsed XML file.
The AXSI is not interested in a documents physical structure.

The cons somewhat outweighed the pros when deciding that this technology is not appropriate to implement in AXSI. Most importantly is the fact that AXSI is not interested in an XML Schema definition document structure but rather the data that the document constraints. Hence, the DOM contains an unnecessary amount of information kept in memory that perhaps the AXSI will never modify or never even use. [16]

5.5.2. Using the XSLT API

The XSLT API defined in the javax.xml.transform package lets users transform XML into different formats with a transformation process. A source object is the input to the transformation process. A SAX or DOM reader or some other input stream can act as a source. Similarly, the result object is the result of the transformation process. That object can be a SAX event handler, a DOM, or an output stream.

A set of transformation instructions, defined in a style sheet, can specify how the transformer should format for the output. For instance, the transformation instructions could specify how the transformer should transform an XML source into HTML or a different XML structure.

The XSLT is an interpreted declarative transformation language that also uses an addressing language called XPath to identify nodes in a source. Thus, the XSLT API is very useful for identifying structures within a document and AXSI could use this to identify the primary components in an XML Schema definition as specified in a style sheet when importing an XML Schema definition into the Amos II system. This is very convenient but the problem is how to define the style sheet correctly and what the output should be like. For instance, the output can be a direct translation to a database schema or some intermediate data structures that suite the Amos II system, perhaps a DOM containing a representation of what will be functions and types. Other AXSI modules can then process this DOM in similar way the current architecture works. When AXSI was implemented using this technology, a significant performance decrease occurred, and the reason being, the interpretative nature of the XSLT language.

The pros of using the XSLT APIs with the AXSI
Easily identifies primary components within an XML Schema definition. XSLT creates views over a document with only a few lines of code.

The cons of using the XSLT APIs with the AXSI
Need to know new languages to use XSLT and XPath.
The interpretative nature of XSLT decreases performance.

This technology never was much of an alternative for the AXSI as it turns out. The performance dropped too much and in addition, the several issues with defining the style sheet in a general way to identify the primary components for possibly many different XML Schema definitions showed to be difficult but not impossible. [18]

Chapter 6

Discussion

This chapter raises some issues for discussion. The first section review the achieved result for the thesis, the second section looks at the importance of other research work within the same research area as this thesis, the third section raises the usefulness of the thesis, the fourth section contains the bibliography and finally, the last section contains acknowledgements.

6.1. Achieved result

We described the architecture of AXSI; an XML Schema definition importer tool, which parses an XML Schema definition and translates it into a database schema that an object-oriented database mediator system can use. Further more, the thesis present a set of translation rules for XML Schema definition language structures and mappings of XML Schema definition language datatypes.

6.2. Previous research and results

Some of the issues raised in the paper by H. Lin et al. [23] are quite similar to the problems and solutions in this work. In their paper, they propose a wrapper called AmosXML that uses a parser to read DTD definitions, and then use translation rules to import the semantics of a DTD definition into an object-oriented database mediator system. The similarities between AmosXML and the AXSI is the fact that both use parsers and translation rules, both in their work and in ours fortify the correctness of both solutions.

The Java WSDP [13] showed to be very useful for the implementation of AXSI and especially the JAXP framework [14] since AXSI is XML-enabled with the provided SAX interface of JAXP.

6.3. Usefulness

The thesis proposes how an XML-enabled tool called AXSI can import an XML Schema definition into an object oriented database mediator system. It can import an XML Schema definition with basic XML Schema definition language components but needs further development in order to support a larger number of XML Schema definitions. The tool is currently a prototype that together with the Amos II system helps to solve the integration problem between applications and data expressed in XML that is defined by XML Schema definitions and researchers can use this thesis to find a possible solution to the integration problem.

6.4. References

Most of the references in the bibliography section of the thesis are provided from Internet sources. However, the sources are reliable as most of them are provided by well-known organizations. Several different specifications are used throughout the thesis as references and these are of course subject to change. The implications of this can result in an AXSI implementation that no longer follows the specification.

6.5. Acknowledgements

We thank our supervisor at Uppsala Database Laboratory, professor Tore Risch, for his support and enthusiasm for the project and the layout and contents of this thesis.

Chapter 7

Conclusions and future work

This thesis proves that it is possible to import an XML Schema definition into an object-oriented database mediator system in an obvious and useful way, which reflects the intended meaning of an XML Schema definition. A developed prototype tool called AXSI performs the importation automatically between the data model of an XML Schema definition into the data model used by the Amos II system by using translation rules to translate the semantics of an XML Schema definition and a direct mapping of contained datatypes. The results are validated against a known XML Benchmark called XBench.

7.1. Answers to question at issue

The development of the prototype tool answered the issued questions and this section briefly summarizes the result. The main question was; how an XML Schema definition is imported into an object-oriented database mediator system by using such a schema importation tool? The question was divided further into three sub questions.

Q1. How is a schema importation tool that imports an XML Schema definition into the object-oriented database mediator designed? If the tool is automatic, it can translate different XML Schema definitions automatically. Consequently, an analysis of several translations will reveal if the tool performs the importation correctly or not.

A1. There are many different solutions to this. This work specifies the architecture of a prototype tool called AXSI implemented in Java that uses a SAX parser and the Amos II system’s Java callout interface. The AXSI is able to translate an XML Schema definition into a database schema used by the object-oriented database mediator system Amos II.

Q2. Can the tool translate the structures of an XML Schema definition to the database schema, in an obvious and useful way, which reflects the intended meaning of the XML Schema definition? XML Schema part 1: Structures recommendation shows the structure of the XML Schema definition language and the language contains several different components that govern its structure [4].

A2. Yes, our work shows that the chosen object-oriented database mediator system Amos II has the necessary programming structures and semantics to represent the structure of an XML document defined by an XML Schema definition. AXSI can use translation rules to translate the structures in the XML Schema definition into the data model used by the Amos II system using the system’s programming structures and semantics.

Q3. Can the tool map XML Schema definition language datatypes to corresponding datatypes in the object-oriented database mediator? A study of XML Schema part 2: Datatypes recommendation describes supported datatypes for the XML Schema definition language [5].

A3. Built-in XML Schema definition language datatypes can be mapped, but due to lack of equivalents in the object oriented database mediator system Amos II, some loss of semantics and precision cannot be avoided.

7.2. Further research

The included XML Schema definitions in the XBench benchmark suite cover only a subset of the XML Schema definition language. Other adaptive translation rules should also be defined for other schema components and structures in order to support a larger subset of the XML Schema definition language. Below are some other interesting research areas in which improvement of the integration between object-oriented database mediators and XML could continue.

7.2.1. Data importation

A data population tool can be developed that populates the object-oriented database mediator system with data from an XML instance document, conforming to an XML Schema definition that AXSI has previously imported into the object-oriented database mediator, using only the imported database schema.

7.2.2. Querying

Fully develop a wrapper that uses the XML Schema definition importer tool, a data loader and also adds query capabilities over XML instance documents.

7.2.3. Web Service Interface

Develop a Web Service interfaces to support communication via XML-based interfaces. A wrapper to use with an object-oriented database mediator system could support WSDL, and a first step is taken with AXSI as WSDL prefers to use the XML Schema definition language to express contained types that are used in messages [41]. However, this requires considerable work but AXSI can provide a basis for further research.

7.2.4. Extending Amos II type hierarchy with new literals using Java

The set of literals in the Amos II system contains the most basic datatypes. Is it possible to extend the Amos II type hierarchy, see Figure 1, with datatypes from Java? For example, the Java package java.math provides classes for performing arbitrary-precision integer arithmetic and arbitrary-precision decimal arithmetic (BigInteger and BigDecimal). Can they be built into the system, or perhaps, used through user-defined types that store the values as a charstrings and uses the java interfaces, callin and callout, to perform arithmetic? Naive experiments have actually been conducted for the latter case and they indicated that it could be done. However, more extensive experiments need to be conducted to see whether it is fully possible or not and if it might become a performance issue. By extending the Amos II system with new literal, a more precise datatype mapping between XML Schema definitions and the Amos II system could be done.

Bibliography

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, *Extensible Markup Language (XML) 1.0 (Second Edition).* World Wide Web Consortium (W3C), October 2000, http://www.w3.org/TR/2000/REC-xml-20001006.

[2] W3C, *HTML 4.01 Specification*. World Wide Web Consortium. W3C Recommendation, 24 December 1999. http://www.w3.org/TR/html4/.

[3] W3C, *XML Schema Part 0: Primer*.World Wide Web Consortium. W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-0/.

[4] W3C, *XML Schema Part 1: Structures*. World Wide Web Consortium. W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-1/.

[5] W3C, *XML Schema Part 2: Datatypes*. World Wide Web Consortium. W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-2/.

[6] Eric van der Vlist, *XML Schema*. O'Reilly & Associates, Inc. June 2002 First Edition. ISBN: 0-596-00252-1.

[7] Tore Risch, *AMOS II Active Mediators for Information Integration*. Uppsala Database Laboratory. http://user.it.uu.se/~udbl/amos/amoswhite.html.

[8] Tore Risch, Vanja Josifovski, Timour Katchaounov, *Amos II Concepts*. Uppsala Database Laboratory, June 23, 2000. http://user.it.uu.se/~udbl/amos/doc/amos_concepts.html.

[9] Staffan Flodin, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, and Magnus Werner, *Amos II User's Manual*. Uppsala Database Laboratory, June 23, 2000. Latest revision April 25, 2003. http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html.

[10] Daniel Elin, Tore Risch, *Amos II Java Interfaces*. Uppsala Database Laboratory, Department of Information Science, Uppsala University, Sweden. 2000-08-25.

[11] Gustav Fahl, Tore Risch, *AMOS II Introduction*. Uppsala Database Laboratory, Department of Information Science, Uppsala University, Sweden. October 1, 1999.

[12] Benjamin Bin Yao, M. Tamer Özsu and John Keenleyside, *XBench - A Family of Benchmarks for XML DBMSs*, TR-CS-2002-39, University of Waterloo, December 2002. http://db.uwaterloo.ca/~ddbms/publications/xml/TR-CS-2002-39.pdf.

[13] E. Armstrong et al, *The Java Web Services Tutorial*. Sun Microsystems, Inc. October 9, 2003. http://java.sun.com/webservices/docs/1.3/tutorial/doc/index.html.

[14] Rajiv Mordani, Scott Boag, *Java API for XML Processing (JAXP) Specification*. Sun Microsystems, Inc. Version 1.2 Final Release, September 6, 2002. http://java.sun.com/xml/docs.html.

[15] Joseph Fialli, Sekhar Vajjhala, *The Java Architecture for XML Binding (JAXB) Specification*. Sun Microsystems, Inc. Final, V1.0, January 8 2003. http://java.sun.com/xml/docs.html.

[16] W3C, *Document Object Model (DOM) Level 2 Core Specification*, Version 1.0, World Wide Web Consortium. W3C Recommendation, 13 November, 2000. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/.

[17] David Brownell, *Simple API for XML*. Saxproject.org. http://www.saxproject.org/.

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

[19] S.Higgins et al, *Oracle 9i XML Database Developer's Guide - Oracle XML DB*. System documentation release 2, Oracle Corp., Redwood Shores, 2002.

[20] Ronald Bourret, *Mapping DTDs to Databases*, O'Reilly XML.com, May 09, 2001. http://www.xml.com/pub/a/2001/05/09/dtdtodbs.html.

[21] Ronald Bourret, *XML and Databases*. Copyright 1999-2003 by Ronald Bourret, Last updated November, 2003. http://www.rpbourret.com/xml/XMLAndDatabases.htm.

[22] Ronald Bourret, et al, *XML-DBMS Middleware for Transferring Data between XML Documents and Relational Databases*. http://www.rpbourret.com/xmldbms/index.htm.

[23] Hui Lin, Tore Risch, Timour Katchaounov, *Adaptive Data Mediation over XML Data*, Uppsala Database Laboratory. http://user.it.uu.se/~torer/publ/jass01.pdf.

[24] R. G. G. Cattell, Douglas K. Barry, Mark Berler, et al, *The Object Data Standard: ODMG 3.0*. Object Data Management Group, January 2000, ISBN: 1-55860-647-5.

[25] Elmarze R, Navathe S.B, *Fundamentals of Database Systems*. Fourth edition, 2003. ISBN: 0-321-20448-4.

[26] G Wiederhold: *Mediators in the Architecture of Future Information Systems*. IEEE Computer, 25(3), 38-49, 1992.

[27] Timour Katchaounov, *Query Processing for Peer Mediator Databases*. Acta Universitatis Upsaliensis. Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 901. 73 pp. Uppsala. ISBN: 91-554-5770-3.

[28] Tore Risch, Vanja Josifovski, *Distributed Mediation by Object-Oriented Mediator Servers*. Dept. of Information Science Uppsala University, Sweden, December 21, 2000. http://www.dis.uu.se/~udbl/publ/concur00.pdf.

[29] P. Lyngbaek et al: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[30] D.Shipman, *The Functional Data Model and the Data Language DAPLEX*, ACM Transactions on Database Systems, 6(1), 1981

[31] ISO 8879:1986(E). *Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*. October 1986, ISO (International Organization for Standardization).

[32] Schmelser, Vandersypen, Blomberg, et al, *XML and Web Services Unleashed*. Sams Publishing, February 2002. ISBN: 0-672-323419.

[33] W3Schools, *XML Schemas - Why?* W3Schools, Refsnes Data, 1999 - 2003 http://www.w3schools.com/schema/schema_why.asp

[34] W3C, *Namespaces in XML*. World Wide Web Consortium W3C, January 14, 1999. http://www.w3.org/TR/REC-xml-names/.

[35] The Michigan Benchmark Team, *The Michigan Benchmark: Towards XML Query Performance Diagnostics*. http://www.eecs.umich.edu/db/mbench.

[36] T. Bohme and E. Rahm, *XMach-1: A Benchmark for XML Data Management*. In Proceedings of German Database Conference BTW2001, Oldenburg, Germany, March 2001. http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html.

[37] A. R. Schmidt, F. Wass, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. *The XML Benchmark Project*. Technical report, CWI, Amsterdam, The Netherlands, April 2001. http://monetdb.cwi.nl/xml/index.html.

[38] S. Bressan and G. Dobbie and Z. Lacroix and M. L. Lee and Y. G. Li and U. Nambiar and B. Wadhwa, *XOO7: Applying OO7 Benchmark to XML Query Processing Tools*. In Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM), Atlanta, Georgia, November 2001. http://www.comp.nus.edu.sg/~ebh/XOO7.html.

[39] Runapongsa Jignesh M. Patel H. V. Jagadish Yun Chen Shurug Al-Khalifa Kanda, *The Michigan Benchmark: Towards XML Query Performance Diagnostics*. Department of Electrical Engineering and Computer Science, The University of Michigan, USA. http://www.eecs.umich.edu/db/mbench/mbench.pdf.

[40] Thierry Violleau, *Java Technology and XML Part 2: API Benchmarks*. Sun Microsystems, Inc. From the Java Developer Service, March 2002. http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML_part2/

[41] W3C, *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium W3C, W3C Note 15 March 2001. http://www.w3.org/TR/wsdl