MASTER OF SCIENCE THESIS

XML to RDBMS

By

Magnus Karlsson
(mka@corus.se)

Stockholm, September 2000

Supervisor:
Torbjörn Ryeng and Peter Monthan
Corus Technologies AB
Birger Jarlsgatan 20, 11434 Stockholm
Email: try@corus.se
pmo@corus.se

Examiner and Supervisor:
Gerald Maguire
KTH Teleinformatics
Electrum 204, 16440 Kista
Email: maguire@it.kth.se

Abstract

The Extensible Markup Language (XML) becomes more and more widespread as nearly all major players on the market today have accepted XML as an industry standard for exchanging information between server based products. Thus thousands of XML dialects have emerged since XML 1.0 became a W3C recommendation in February 1998.
Corus Technologies AB has developed a server-based product called Corus/ALS© (Application Linking System) that makes it possible to connect client systems with different data representations to each other. A relational database model for each of the client systems is created and the translation from one data representation to another is done with stored procedures in the database.
This thesis introduces a solution for how to store and retrieve XML documents in a Relational Database Management System (RDBMS) from any of the XML dialects that has emerged since XML 1.0 became a W3C recommendation.
After a XML document has been stored in the database in a normalized way, the stored procedures in the Corus/ALS© database can be used to transform it to another XML dialect (or another format supported by the Corus/ALS© system). This will make it possible to translate any XML document to any other XML format.
An XML interpreter was implemented and this implementation verified the theories in this thesis.

Table of contents

1 Introduction

1.1 Background

Corus Technologies AB has developed a system called Corus/ALS© (Application Linking system). The purpose of Corus/ALS© is to make information exchange possible between almost any kind of computer products over a computer network. This is commonly called application integration or Enterprise Application Integration (EAI).

Different applications that need to share information can have different internal data representation, different communications mechanisms and even lack the possibility to communicate. These are problems that the Corus/ALS© system is designed to solve. Since many of the new server products on the market today uses the Extensible Markup Language (XML) to exchange information with other servers there is also a need for Corus/ALS© to be able to understand XML and translate any XML dialect into another known format, EDI, another XML dialect or maybe putting the data directly into a Relational Database Management System (RDBMS).

1.2 Purpose

At the heart of Corus ALS is an Oracle RDBMS and the systems that shall be linked to each other are described in this RDBMS by database tables, columns, etc. The actual translation of data from one system's data representation to another is done in the RDBMS with stored procedures. This makes it possible to integrate different systems with each other no matter what kind of format they use to exchange information with as long as there is a way of getting the data into the Corus ALS RDBMS.
The purpose of the thesis is therefore to investigate if there is a way to interpret and analyze any kind of XML document and make an intelligent decision of what kind of RDBMS data model should be created in the Corus/ALS© RDBMS for that XML dialect. A method to put subsequent messages of this type into the data model that was created should also be a result of analyzing the XML document. The result will therefore be a method to understand and integrate any of the thousands of XML dialects/formats that exist today.
If it is possible to analyse XML documents in this way then the purpose is to design and construct a XML interpreter that is capable of analyzing a XML Document Type Definition (DTD) and creating a Relational database model for that DTD.
Furthermore, the interpreter must be able to store XML documents into that model as well as extract database information as XML documents.

1.3 Constraints

1 The interpreter shall be configurable from information in a database repository or a XML document.
2 Using the XML document's DTD at hand together with the configuration information, the interpreter shall be able to create a relational data model capable of storing all information in the XML document.
3 The interpreter shall, after the relevant model is created, be able to parse XML documents and store data in the database as well as retrieve data from the database and render a XML document.
4 The interpreter shall be able to handle the scenarios of creation, change, and deletion of data in the database.
5 The current W3C work on XML schemas shall be regarded in implementation of the interpreter and definition of configuration info.
6 The coding language should be java and any user interface should be accessible from a web browser.

1.4 Structure of the report

Chapter 2 gives a brief introduction to the W3C XML standards that have emerged over the years.

Chapter 3 discusses the implementation of the interpreter and the theories that it is built upon.

Chapter 4 discusses the design issues and the choices made to accomplish the requirements that were put up before the work began.

Chapter 5 concludes the work that has been done.

Chapter 6 addresses future improvements.

2 XML Basics

2.1 XML 1.0

Back in 1996 the W3C started the work on XML. This work resulted in XML 1.0 wich became a W3C recommendation in February 1998 [1]. It is upon this W3C recommendation that most of the XML enabled applications of today is built. XML 1.0 has its origins in the specifications of the Standard General Markup Language (SGML) language and this is a part of its widespread popularity.
XML is a self-describing language that uses a simple standard way of delimiting text data. The delimiters, or “tags”, are called elements and elements can have attributes that further describe the data they contain. Elements in turn can contain both data and other elements, making it simple to describe metadata along with actual data when creating a XML message. Figure 2-1 shows a hypothetical XML 1.0 message that could be used by an e-business application. The message contains both elements, nested elements and elements with attributes. A XML document is said to be *well-formed* if it conforms to the rules of the XML 1.0 recommendation.

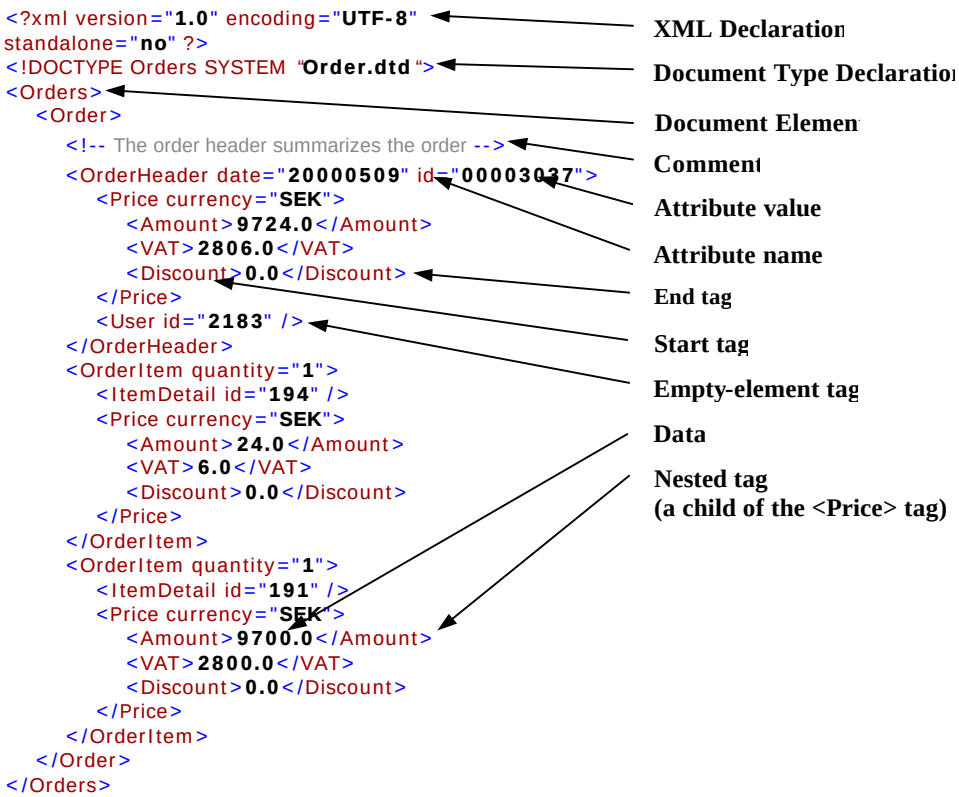


Figure 2-1 A XML 1.0 message with tags and attributes

Naturally there will be a need to communicate the structure of a XML document to another party, as well as communicating the document itself, so that the other party can interpret documents properly. XML 1.0 [1] provides this kind of mechanism as a part of the specification through the use of a Document Type Definition (DTD). The DTD describes the vocabulary of a certain XML dialect. Thus, if a DTD exists, the parser will know what element follows another element and what attributes a certain element may have. Figure 2-2 shows the DTD of the hypothetical XML document in Figure 2-1.

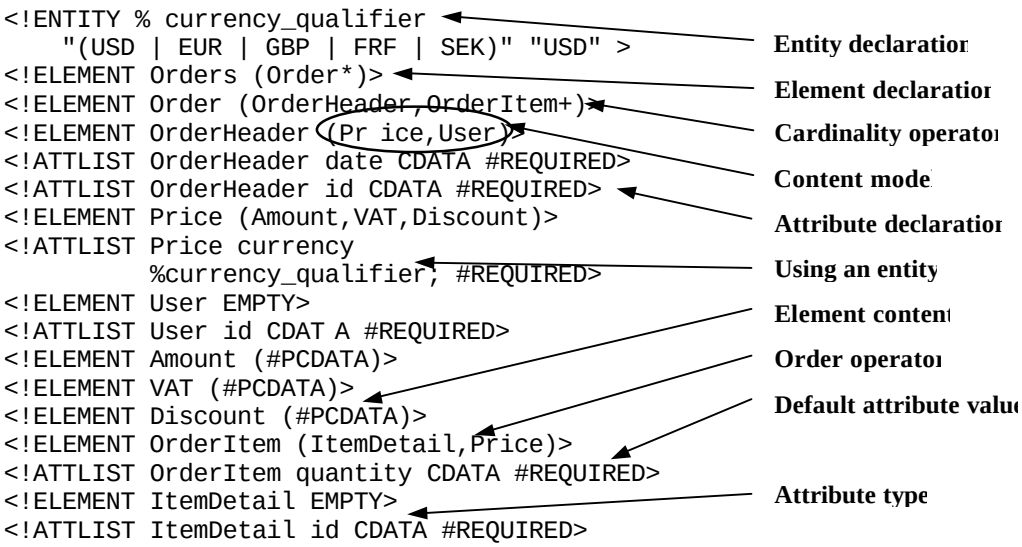


Figure 2-2 A XML 1.0 DTD with element, attribute and entity declarations

A document is said to be *valid* if it conforms to a certain DTD.

2.1.1 XML 1.0 structure

As seen in Figure 2-1 a XML message consists of several tags and some of them are even compulsory according to the specification.
The first part of the document is called the *prolog*. The prolog consists of the *XML Declaration* and the *Document Type Declaration*.
The XML Declaration, which is compulsory in every XML document, has three attributes defined by the XML 1.0 specification:

- ?? version – must be “1.0”. This attribute is compulsory.
- ?? encoding – a legal character encoding such as “UTF-8” or “UTF-16”. This attribute is optional.
- ?? standalone – is either “yes” or “no” and tells the parser if this XML document must be compared to an external DTD or not. This attribute is optional and if left out the implied value is “no”.

The Document Type Declaration is optional and will follow the XML Declaration if it exists. If it exists, then it contains an internal subset of the DTD or refers to an

external subset of the DTD. The Document Type Declaration in Figure 2-1 for example, refer to a DTD that's named "Order.dtd" and can be found in the same directory as the XML document itself since no absolute path is used. It is however possible to use a URL to refer to a DTD as well.
After the prolog comes the *body* of the XML document. The body contains the tags of this particular XML dialect. The first element in the body is the Document Element and this element will in turn contain all other elements of this document. Elements that are immediate children of another element are *nested* elements of that element and thus are all elements in a XML document except for the Document Element nested elements.
Each element can also have attributes. An attribute consists of an attribute name and an attribute value and an element can only have one instance of an attribute name.
An element that does not contain any information at all is called an empty element and consists only of an Empty-element tag and possibly a set of attributes. If an element has content, it will be found between the element's start tag and end tag. The content of an element can be other elements or data or a mix of both data and elements.

2.1.2 XML 1.0 DTD

The DTD is a part of every *valid* XML document. A DTD can be used by any *validating* parser to examine if a valid XML document conforms to the DTD it refers to.
Figure 2-2 shows the DTD of the valid XML document in Figure 2-1.
The DTD in Figure 2-2 consists of three out of four possible constructs. The possible constructs are:

ELEMENT	a declaration of an element.
ATTLIST	a declaration of an attribute.
ENTITY	a declaration of some reusable content.
NOTATION	a declaration of some external content not meant to be parsed. And a reference to the application that handles the content.

The content of an element falls into one of four categories: *empty*, *element*, *mixed*, and *any*. Figure 2-3 shows examples of element declarations that belong to the different categories.
If an element is declared to be empty the element cannot contain elements or data. If the element's content is declared to be of the any type, the element can contain any data or any elements in any order at all. Since declaring the content of an element to be of the any type doesn't say anything about the content of the element to the parser, it is rarely used.
An element is declared to be of element or mixed type by the use of a *content model*, see Figure 2-2. A content model is a set of parentheses that includes child element names, operators, and the #PCDATA keyword.
If the content model starts with the #PCDATA keyword, the element's content is considered to be mixed according to the XML 1.0 specification [1]. Figure 2-3 has an example of an element declaration with a mixed element content indicating that the element can contain a mixture of data and <Price> elements.

If the content model starts with a child element name the element content is of the element type. An element with this kind of content model cannot contain any data, only other elements.

Element Declaration	Element content
<!ELEMENT EmptyElement EMPTY>	Empty
<!ELEMENT AnyInformation ANY>	Any
<!ELEMENT FruitBasket (Apples,Bananas,Grapes)>	Element
<!ELEMENT MixedInformation (#PCDATA | Price)>	Mixed

Figure 2-3 An example of element declarations for different element content

In the content model the child element names are separated by an *order operator*. There are two possible types of order operators; the comma operator "," and the pipe operator "|". The comma operator describes a strict sequence of elements whereas the pipe operator describes a choice of elements. Figure 2-3 shows an example of the use of both a comma operator and a pipe operator.
Content models may themselves be nested to allow more complex structures, as seen in Figure 2-4.

```
<!ELEMENT BigFruitBasket (Apples,(Bananas | Grapes))>
```

Figure 2-4 A nested content model

It is also possible to describe cardinality, i.e. how many child elements of a certain type that is permitted. Cardinality is described thru the use of cardinality operators next to the child element names or next to a content model, as seen in Figure 2-5.

```
<!ELEMENT BigFruitBasket (Apples?,(Bananas | Grapes)+)>
```

Figure 2-5 The use of cardinality operators

There are three different cardinality operators that can be used; the optional operator "?", the zero or more operator "*" and the one or more operator "+". The optional operator is used when a child element or a content model is optional. The zero or more operator is used when a child element or a content model can appear zero or more times and the one or more operator is used when a child element or a content model can appear one or more times.

All the attributes that belong to an element are declared through one or more attribute declarations. An attribute declaration starts with the ATTLIST keyword followed by the name of the element the attribute belongs to, followed by zero or more attribute definitions as can be seen in Figure 2-6. Each attribute definition consists of the name of the attribute, its type, and a default declaration.

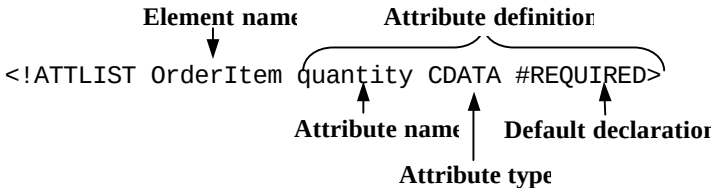


Figure 2-6 An attribute declaration

There are a number of different attribute types that can be used such as: CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS and NOTATION. These different attributes types all imply some sort of restriction of the value an attribute can have. It is also possible to restrict the values of an attribute to a certain series of values. The different attribute types are further described in Figure 2-8.
The default declaration is used to tell whether or not the attribute must occur and if it has a default value. There are four possible combinations for the default declaration as shown in Figure 2-7.

Default declaration	Description
#REQUIRED	The attribute must appear on every element it's declared for.
#IMPLIED	The occurrence of the attribute is optional for the element it's declared for.
#FIXED plus default value	The value of the attribute must always be the default value supplied.
Default value	The value of the attribute will be the default value supplied if no other value is explicitly supplied.

Figure 2-7 The four possible default declarations

Attribute type	Description
CDATA	Character data. The value of the attribute is a string of any length.
ID	A unique value. The value of the attribute must be unique amongst all other attributes of the ID type in the document. The attribute must also be declared #IMPLIED or #REQUIRED.
IDREF	A reference to an element that has an ID attribute with the same value as this IDREF attribute.
IDREFS	A series of references, separated by white space, to elements that have an ID attribute with the same value as one of the valu es in this series.
ENTITY	The value of the attribute will be taken from a predefined entity declared somewhere else in the DTD.
ENTITIES	The value of the attribute will be taken from several predefined entities and the entities will be separated by white space.
NMTOKEN	A NMTOKEN is one or more *NameChar* characters as defined in section 2.3 of the XML 1.0 specification [1]. The parser will delete leading and trailing space for this type of attribute.
NMTOKENS	A series of NMTOKEN, separated by white space. The parser will delete sequences of space.
NOTATION	A NOTATION attribute is used to refer to an external handler to handle data that the XML parser cannot deal with, for example binary data. The actual NOTATION declaration will be found elsewhere in the DTD and it wi ll refer to the external application that will handle the content.
[Enumerated value]	A series of predefined values separated by the pipe symbol (|), which are acceptable as values for the attribute.

Figure 2-8 The different attribute types

2.2 XML Schema

As XML 1.0 became accepted and widespread, developers stared to realize it could be improved in some areas. A strong typing, the ability to validate a document across

multiple namespaces and the use of XML syntax in the DTD were a few of the improvements that seemed obvious.
The DTD in XML 1.0 that we have seen earlier (in section 2.1.2 for example is written) in a syntax called Extended Backus Naur Form (EBNF). Since EBNF has a flat structure, unlike the hierarchical structure of XML, it can be difficult to understand and parse. The Document Object Model (DOM) for example cannot be used to parse the DTD because of this flat structure. If the DTD had been written in XML itself the DOM familiar to every developer experienced in XML could be used to parse the DTD.
Since XML 1.0 became a W3C recommendation before the work on XML namespaces began, namespaces cannot be used in the DTD itself. This means that a DTD cannot be created by using parts of other DTD's.
One of the greatest disadvantages of the XML 1.0 DTD is probably that it does not support data types. The data in a XML document will be treated as text by the parser leaving it up to the programmer to convert the text to other data types where suitable. This is not a big problem if the XML dialect that is used is well known to the application but if there is a need to exchange information amongst applications with different XML dialects, it could pose a problem since there is no way of knowing how to convert data from one XML dialect to another just by looking at the DTD.
Another disadvantage with the DTD in XML 1.0 is that it doesn't allow inheritance. A DTD cannot inherit declarations from another DTD.
Thus there is a need for a new XML standard. The new standard that W3C is currently developing is called XML Schema. W3C has published the specifications for the latest working draft of XML Schema on its website. The working draft is divided in three documents, XML Schema Part 0: Primer [5], XML Schema Part 1: Structures [6], and XML Schema Part 2: Datatypes [7].
Since XML Schema is still under development, I will not delve into the details of how it is built up, although Figure 2-9 shows an example of a XSD (XML Schema Definition language), the equivalent of an XML 1.0 DTD, which can be used to create the XML document in Figure 2-1. As can be seen in Figure 2-9, the XSD itself is defined an XML markup and can thus be parsed by a DOM parser that understands the XML Schema definition. Data types are extensively used in the example as well.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSche  ma">
  <xsd:element name="Orders" type="OrdersType"/>

  <xsd:complexType name="OrdersType">
    <xsd:element name="Order" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:element name="OrderHeader" type="OrderHeaderType"/>
        <xsd:element name="OrderItem" type="OrderItemType"
maxOccurs="unbounded"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>

  <xsd:complexType name="OrderHeaderType">
    <xsd:attribute name="date" type="xsd:date">
    <xsd:attribu te name="id" type="xsd:int">
    <xsd:element name="Price" type="PriceType"/>
    <xsd:element name="User">
      <xsd:attribute name="id">
        <xsd:simpleType base="xsd:positiveInteger">
          <xsd:maxExclusive value="9999"/>
        </xsd:simpleT ype>
      </xsd:attribute>
    </xsd:element>
  </xsd:complexType>

  <xsd:complexType name="OrderItemType">
    <xsd:attribute name="quantity" type="xsd:positiveInteger"/>
    <xsd:element name="ItemDetail">
      <xsd:attribute name="id">
        <xsd:s impleType base="xsd:positiveInteger">
          <xsd:maxExclusive value="9999"/>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:element>
    <xsd:element name="Price" type="PriceType"/>
  </xsd:complexType>

  <xsd:complexType name="PriceType ">
    <xsd:attribute name="currency" type="CurrencyType" value=”USD”/>
    <xsd:element name="Amount" type="xsd:decimal"/>
    <xsd:element name="VAT" type="xsd:decimal"/>
    <xsd:element name="Discount" type="xsd:decimal"/>
  </xsd:complexType>

  <xsd: simpleType name="CurrencyType" base="xsd:string">
    <xsd:enumeration value="USD"/>
    <xsd:enumeration value="EUR"/>
    <xsd:enumeration value="GBP"/>
    <xsd:enumeration value="FRF"/>
    <xsd:enumeration value="SEK"/>
  </xsd:simpleType>

</xsd:sche ma>
```

This is the Document Element of this XSD. All other elements are children of this elemen

Data types are a part of the XML Schema definition

It is possible to limit the range of a data type

Figure 2-9 The XSD of the XML document inFigure 2-1

2.3 DOM

The Document Object Model (DOM) is a programming interface that can be used by programs and scripts to read and manipulate XML documents. The DOM interface is defined by the W3C but the W3C has not made an implementation of the interface itself. The actual implementation of the DOM interface is left up to the companies that are interested. Since the work of W3C has such an impact on the Internet community almost every company that has made a XML parser has implemented the DOM interface. Companies and organizations such as Microsoft, IBM, Oracle and the Apache Software Foundation have all made implementations of the DOM interfaces. When DOM is used to manipulate a XML document it builds a tree representation of the XML document in memory. The nodes of the tree can then be read, changed or deleted. When a parser has created the DOM tree it gives the caller a handle or a pointer to the root node. The root node represents the Document Element. All other nodes in the tree will be children, grand children etc. to the root element. The tree is traversed through methods in the DOM interface that can get the children of any node that the program happens to have a pointer to.
Element, element content, attributes and text are all nodes in the DOM tree although of different node types.

2.3.1 DOM Level 1

The fist version of the Document Object Model, DOM Level 1[11], became a W3C Recommendation in October 1998. DOM Level 1 defines the following node types:

Document	The entire XML document.
DocumentFragment	A portion of a XML document.
DocumentType	An interface to the list of entities that are defined for the document.
EntityReference	The reference to an entity. Can be used to create a reference to an entity as well.
Element	An element in the XML document.
Attr	An attribute in the XML document.
ProcessingInstruction	A processing instruction in the XML document.
Comment	A comment in the XML document.
Text	Text in the XML document.
CDATASection	Text that would be regarded as markup if not declared as CDATA.
Entity	An entity in the XML document.
Notation	A notation declared in the DTD.

Nodes of a certain type can have nodes of other types as children. The structure that is outlined in the specification is described in Figure 2-10.

Node type	Child node types
Document	Element, ProcessingInstruction, Comment, and DocumentType
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, and EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, and EntityReference
Element	Element, ProcessingInstruction, Comment, Text, CDATASection, and EntityReference
Attr	Text and EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, and EntityReference
Notation	No children

Figure 2-10 The hierarchy of the node types in the DOM tree

As can be seen, all different declarations in the XML 1.0 DTD have their counterparts in the DOM tree, which was the intention when DOM Level 1 was created.

2.3.2 DOM Level 2

When the DOM Level 1 specification was created, Namespaces and style sheets did not exist, so now that both Namespaces and style sheets have reached W3C recommendation, a new version of the Document Object Model is needed. The new version is called DOM Level 2 [12] and has at the time of writing the status of Candidate Recommendation. A Candidate Recommendation is the last stage before an actual Recommendation. It means that W3C is waiting for other parties to do implementations of the interfaces and return with technical feedback before deciding if the specification is complete enough to become a W3C Recommendation.

The DOM Level 2 specification builds upon the DOM Level 1 specification so all interfaces from the DOM Level 1 specification still exist in the new DOM Level 2 specification. DOM Level 2 adds the following to the old specification:

- ?? Support for Namespaces so that existing namespaces can be interrogated and new namespaces created.
- ?? Support for style sheets so that style sheets can be queried and manipulated through a separate object model.
- ?? A built in event model that makes it possible to register event handlers for events caused by user interaction, logical events or events caused by a modification of the structure of the document.
- ?? A range interface that makes it possible to refer to a set of nodes as a range.
- ?? An interface for filtering and traversing a document's content.

2.4 SAX

The Simple API for XML (SAX) is a programming interface which can be used by programs and scripts to read XML documents. SAX cannot be used to create a XML document, like the DOM interface.
SAX is an event-based interface that can be implemented by a XML parser. A XML parser that has implemented the SAX interface will notify the application with a stream of parsing events as it reads the XML document. A parsing event is for example a notification to the program that the parser encountered the start tag of a certain element. The parser will not build an in memory representation of the document so it will be up to the application to buffer data or build its own in memory representation of the data that the parser reads if there is a need to go back to a previous element.
The obvious benefits of an event-based interface are speed and memory efficiency since the parser doesn't need to build an in memory representation of the document. This also means that the SAX interface can be used to parse files of any size. If however an application that uses the SAX interface builds its own in memory representation of the entire document it might be just as inefficient as if a DOM interface would have been used.

2.4.1 SAX v1.0

The first version of SAX, SAX v1.0 [13], was released in May 1998. The work was lead by David Megginson and all of the discussions took place on the public mailing list XML-DEV. Today the SAX interface is supported by virtually every Java XML parser.
When using a SAX v1.0 parser, an application registers itself as the receiver of the parsing events from the parser. The application then implements code to take care of different events from the parser. The events that a SAX parser can send to an application are shown in Figure 2-11.

Event	Passed parameters
Start of the document	No parameters passed
End of the document	No parameters passed
Start of an element	The element name and all the attributes
End of an element	The element name
Character data	A character array with the content of an element
White space separating elements	A character array with the spaces, tabs and newlines
A processing instruction	A target name and arbitrary character data

Figure 2-11 The type of eventsthat a SAX v1.0 parser can send to an application

2.4.2 SAX v2.0

The SAX v2.0 interface [14] was created to address some of the limitations that the SAX v1.0 interface has. The limitations that have been addressed is support for namespaces, support for parsing the DTD and the addition of interfaces for access to the boundaries of internal entities, the boundaries of CDATA sections and the existence of comments.
The specification was released in its final version in May 2000 and at the time of writing only three parsers have implemented the full specification according to the official SAX v2.0 web page [14]. The three parsers are:
The Apache Software Foundation's Xerces Java Parser, David Brownell's SAX2 XML Utilities and Michael Kay's SAXON.

2.5 XSL

The Extensible Stylesheet Language [8] (XSL) is an XML based language for expressing style sheets. XSL style sheets can be used to transform a XML document into another XML document.
During the development of XSL it became clear that the language consisted of two parts: one part describing the vocabulary or the XML dialect used and one part describing the structural transformation, in which element are selected. The two specifications are: the Extensible Stylesheet Language [8] (XSL) describing the XML vocabulary used and the XSL Transformation [2] (XSLT) specification describing the transformation language.
As the work proceded, it was recognized that there was a need for a way of selecting parts of a document. At the same time the W3C was developing the XML Pointer language (XPointer) to be used for linking from one document to another and they

also needed this functionality. Thus the two comities joined forces and defined a new language: the XML Path Language [4] (Xpath) describing a way of addressing a part of a document.
The XSL language is still under development but the two sub standards XSLT and Xpath reached W3C recommendation in November 1999. XSL has wonderful facilities for achieving high-quality typographical output but in this thesis we are more interested in transforming XML documents. XSLT can in fact also be used to generate formatted output since it can be used to generate HTML and Cascading Style Sheet [10] (CSS or CSS2) output.

2.5.1 XSL Transformations (XSLT)

XSL Transformations [2] (XSLT) reached the status of W3C recommendation in November 1999. It is a tool for transforming XML documents.
XML Namespaces are considered to be an essential part of the XSLT language and this is taken into consideration for all XML documents that are transformed.
When XSLT is used to transform a XML document, a XSLT processor is used. The XSLT processor builds an internal model called a tree for the source document and the style sheet and uses the style sheet tree to transform the source tree into a result tree. The result tree is then be used to create the result document. The output can be xml, html, or text.
The XSLT style sheet uses XML tags from the XSLT Namespace to give instructions to the XSLT processor. XML tags from any other Namespace will not be regarded as instructions to the XSLT processor and will be copied to the result document.
The XSLT Namespace is declared in the root element of the style sheet and the declaration looks like this: xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”. Thus will all tags that start with “xsl:” will be regarded as instructions to the XSLT processor.
One of the most used XSLT instructions is the template rule. A template rule is expressed in the style sheet as an <xsl:template> element with a match attribute. The value of the match attribute is a pattern. The pattern determines which of the nodes in the source tree the template rule matches. For example, the pattern “/” matches the root node “Order/OrderHeader” matches the <OrderHeader> element which is the child of the <Order> element. It is pattern like this one that the Xpath language is used for.
When the XSLT processor parses the source document it will start with the root element and look for a corresponding XSLT template rule in the style sheet document. If this template rule is found, the XSLT instructions in this template rule will be carried out. A template rule can contain XSLT elements that will make the XSLT processor call template rules for the child elements to the element in focus in the source document and in this manner the source document can be traversed and the right output created using several template rules.
The different XSLT elements that can be used in a style sheet are:

<xsl:template>	The template rule.
<xsl:apply-templates>	Used to call one or several template rules.
<xsl:call-template>	Used to call a single template rule.
<xsl:stylesheet>	The root element of the style sheet.

<xsl:include>	Used to include the content of a style sheet into another style sheet.
<xsl:import>	Used in the same way as the <xsl:include> element but the definitions in the imported style sheet will be used in preference to those that already exist.
<xsl:value-of>	Writes the string value of an expression to the result tree.
<xsl:attribute>	Creates an attribute to an element.
<xsl:element>	Creates an element.
<xsl:comment>	Creates a comment.
<xsl:processing-instruction>	Creates a processing instruction.
<xsl:text>	Creates literal text.
<xsl:variable>	Declares a local or global variable that can be used by the XSLT processor.
<xsl:param>	Declares a parameter that can be used to pass a data.
<xsl:with-param>	Used to set the value of a <xsl:param>.
<xsl:copy>	Copies the current node in the source document to the current output destination.
<xsl:copy-of>	As <xsl:copy> but copies all descendant nodes to.
<xsl:if>	As any ordinary if statement but in XSLT.
<xsl:choose>	Works like a switch statement.
<xsl:when>	The condition to be tested inside a <xsl:choose> element.
<xsl:otherwise>	Used if all <xsl:when> conditions failed inside a <xsl:choose> element.
<xsl:for-each>	Selects a set of nodes and performs the same processing for all of them.
<xsl:sort>	Used to specify the order in which nodes are selected by the <xsl:apply-templates> or <xsl:for-each>.
<xsl:number>	Used to allocate a sequential number or to format a number for output.
<xsl:output>	Used to control the format of the output from the XSLT processor.

An example of a style sheet that can be used to transform the XML document in Figure 2-1 into the very simple XML document in Figure 2-13 is shown in Figure 2-12.

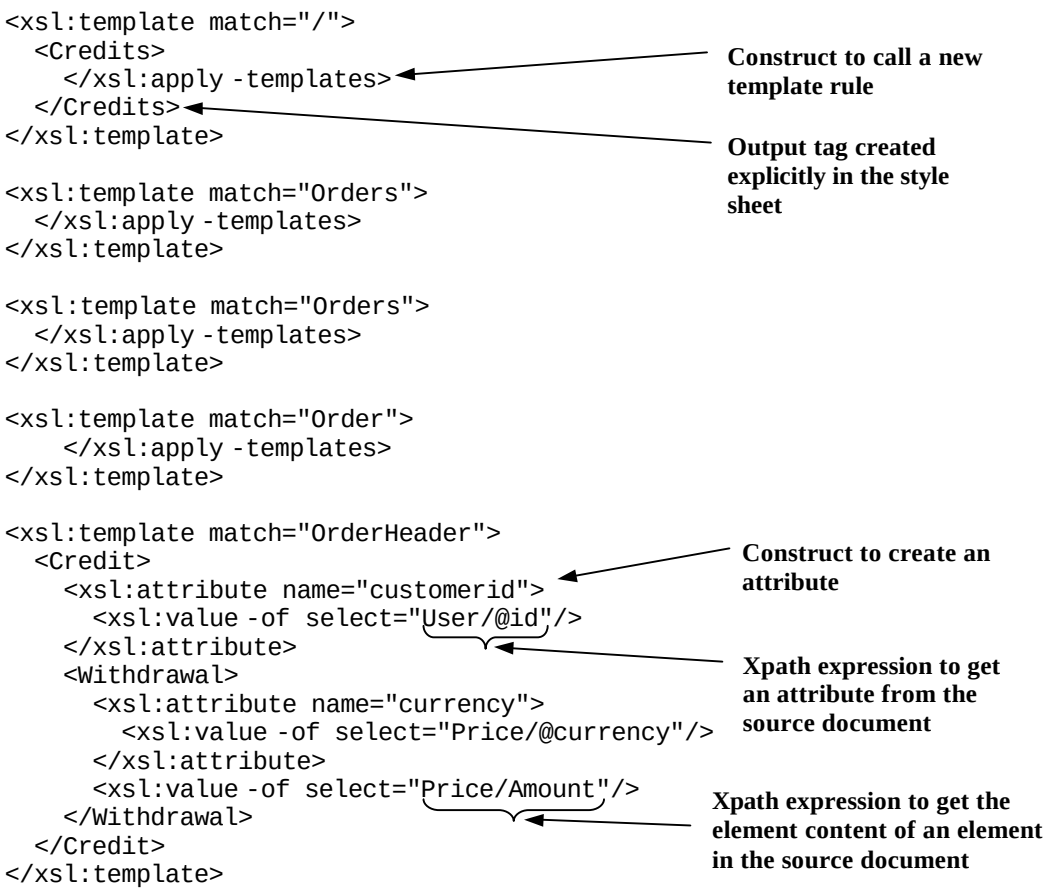


Figure 2-12 A sample XSLT style sheet

```
<Credits>
  <Credit customerid="2183">
    <Withdrawal currency="SEK">9724.0</Withdrawal>
  </Credit>
</Credits>
```

Figure 2-13 The output from the style sheet in Figure 2-12

2.5.2 XML Path Language (Xpath)

The XML Path Language [4] (Xpath) reached the status of W3C recommendation at the same time as the XSLT language in November 1999. The primary purpose of Xpath is to address parts of a XML document. Basic facilities for manipulation of strings, numbers and booleans are also a part of the Xpath language.
Nodes in a XML document can be addressed using a *location path*. The location path starts with an *axis*. The axis is used to define the type of node that should be selected. An example of the use of an axis is “child::para”, where “child” is the axis used and “para” is the element that will be selected. There is also an abbreviated way of using a location path where axis can be omitted. The abbreviated form when using the “child::” axis is simply to omit it the axis declaration.

The following axis's can be used:

ancestor	Selects all the nodes that are ancestors to the currently selected node, with the parent as the first node and the document root as the last node.
ancestor-or-self	Same as the ancestor axis but with the currently selected node as the first node.
attribute	Selects all the attributes of the currently selected node.
child	Selects all the children of the currently selected node.
descendant	Selects all children, children's children etc. from the currently selected node and downwards.
descendant-of-self	Same as the descendant axis but with the addition of the currently selected node as the first node.
following	Selects all nodes that follow the currently selected node in the document.
following-sibling	Selects all nodes that has the same parent as the currently selected node and is following the current node in the document.
namespace	Select all namespace nodes that are in use by the currently selected node.
parent	Selects the parent node to the currently selected node.
preceding	Selects all nodes that precede the currently selected node in the document.
preceding-sibling	Selects all nodes that has the same parent as the currently selected node and are preceding the current node in the document.
self	Selects the currently selected node.

The most commonly used way of selecting nodes are by using the abbreviated form though.
Location paths can also be either relative or absolute. A relative location path simply means that nodes are selected by giving the position of the nodes relative to the node that is currently selected while an absolute path means the position of a node relative to the document root. For example: "//Orders/Order/OrderHeader" is an example of an absolute location path in abbreviated form that selects the OrderHeader element.

2.6 Namespaces in XML

Namespaces in XML reach the status of W3C recommendation in January of 1999. XML Namespaces were created for the purpose of solving the problems with ambiguity and name collisions that existed with XML 1.0 if multiple DTD's were to be used for the same XML dialect. The problems arise when different DTD's have different declarations for the same constructs. If, for example an element was declared as empty by one DTD and another DTD declared it to have children these two declarations would be in contradiction to each other and it would be impossible to know which one of the two declarations that should be used. The solution to the problem is to group all elements and attributes declared in the same DTD together and

then tell the elements and attributes apart by looking at what group they belong to. A group or collection is identified by a namespace declaration that uses a Uniform Resource Identifier (URI) to give the resource a unique name. There are two ways of using the URI when declaring a namespace, either by using an urn or a HTTP location:
xmlns=”http://www.corus.se/xml/sales/sales.dtd”
xmlns=”urn:corus-sales-stock-stockdefs”
“xmlns” is a reserved word from the Namespace recommendation and cannot be used for any other purposes.
A XML document can be declared to have a default namespace and if it has a default namespace all elements and attributes that doesn’t have a *qualified name* will be part of that namespace. An *alias* is provided for a namespace declaration to make it possible to refer to it using the qualified name. Here are the two previous declarations with an alias:
xmlns:sales=”http://www.corus.se/xml/sales/sales.dtd”
xmlns:stock=”urn:corus-sales-stock-stockdefs”
These two namespace aliases can then be used by their qualified name:
<stock:item sales:price=”10”>

2.7 XML Parsers

No one knows how many different XML parsers exist but a qualified guess would be more than 50. A list of about 40 of them can be found at http://www.xmlsoftware.com/parsers/.
The question of which parser to choose depends on what environment it will run under, what XML interfaces it implements and what kind of support the different vendors can give. One of the most interesting parsers today is the one developed under the Apache XML Project, an open source initiative that can be found at http://www.apache.org. The Apache XML Project XML parser, called Xerces, is based on the Suns Crimson parser that Sun has given to the Apache XML Project. Xerces is available in both Java and C++ and has support for both DOM Level 1 and Level 2 and SAX version 2. The Xerces parser is also attractive because it supports 24 different character encodings.
If an Oracle database is used then it could be interesting to use the Oracle parser implementations. The Oracle parsers has implemented the DOM Level 1 and SAX version 1 interfaces and has support for 15 different character encodings. The Oracle parsers exist in Java, C, C++, and PL/SQL.

3 The XML Interpreter

This chapter discusses how the XML interpreter was designed to fulfill the predefined requirements.
The actual implementation utilizes the Oracle XML Developer's kit (XDK) for Java, which can be freely obtained from Oracle. The Oracle XDK contains a XML parser and a XSLT processor that is used in the implementation.
I chose the Oracle parser because Oracle has implemened the Java Runtime Environment (jre) into the actual database and is planning to implement a servlet engine into the actual database in its next version. Since the Corus/ALS© system uses an Oracle database this could potentially mean significant performance gains.
The XML Schema is still a working draft and many of the XML parsers doesn't support XML Schema, even the ones that do are in early alpha versions which only support subsets of different versions of the working draft. Because of this, the decision fell upon XML 1.0 for the actual implementation of the interpreter. Thus, when the working draft reaches the recommendation stage, a new version of the interpreter will need to be implemented.

3.1 The design of the interpreter

The design goals of the interpreter were:

- ?? The interpreter should be configurable from information in a database repository or a XML document.
- ?? Using the XML document's DTD, the interpreter should be able to create a relational data model capable of storing all information in the XML document.
- ?? After the relevant model is created, the interpreter should be able to parse XML documents and store data in the database as well as retrieve data from the database and render a XML document.
- ?? The interpreter should be able to handle the scenarios of creation, changes, and deletion of data in the database.
- ?? The coding language should be java and any user interface should be accessible from a web browser.

These goals implie a design of the interpreter where the interpreter first parses a XML document's DTD and from that DTD generates one or several configuration files that could be used later to:

- ?? Create the relational data model that would be able to store all future XML documents of that type.
- ?? Take the XML document, and all future XML documents of this type, and put its data into the data model.
- ?? Extract data from the relational data model and recreate a XML document of that type.

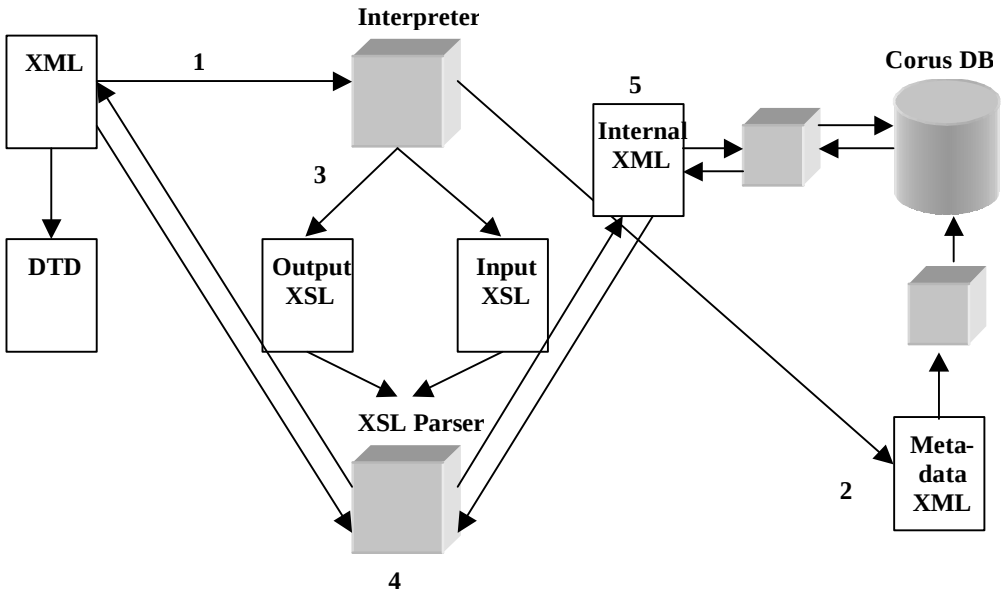
To be able to create the relational data model from a configuration file a special metadata XML format was created. This metadata XML dialect is further discussed in section 3.2.
When it comes to storing the XML document in the database model that has been created there are a number of different approaches that can be used. They all share common communication mechanisms and by knowing either where the XML document is sent or fetched from or by parsing the root tags of the XML document, it is possible to find out which database model should be used.

One approach is to have the interpreter examine the structure of each XML document it encounters and compare it to the database model at hand and have the interpreter make decisions of what data it should put into what column and table in the database model. However this approach fails on a number of points. First of all it is very inefficient since the XML document's structure needs to be examined every time there is a new XML document to parse. Secondly it may be impossible for the interpreter to know what data in the XML document should be put into a certain database column since column names are decided by the user. Thus a more sophisticated approach is needed.

If the interpreter was to create a description of how the mapping of data in the XML document to the relational database model was done when creating the database model, then that description could be used when an instance of a XML document was to be inserted into the database model. The description must contain information of how the XML document should be divided during the parsing to be able to put data from different parts of the XML document into different database tables, how the elements in the XML document map to the columns in the tables and what relations exist between the different database tables and how they should be created. This means that the interpreter will be very complicated and advanced in order to be able to parse any XML dialect using the description discussed and this means that performance could be a problem. Performance and maintenance of the interpreter are essential if the interpreter is going to be a part of the Corus/ALS© system since it must be able to parse documents from tens or even hundreds of sources at once.
A different approach that could meet the demands was thus desirable.

After examining the work done by the W3C in the area of transforming XML documents [2], literature from Wrox on XSLT [16], and also the work done by Microsoft in their BizTalk server (that can be downloaded for free in a beta version from www.biztalk.org) a new approach surfaced.
The selected approach uses XSL Transformations to convert a XML document of a certain dialect into a XML document that conforms to a XML format that is natively used by the Corus/ALS© system. When inserting data from the native XML format into the Corus/ALS© database there is no need for a description file because the native XML format is self- describing and contains all necessary information itself. The connection to the database from this internal XML format can thus be hard coded and will be very fast and small in size. The conversion from the external XML format to the internal XML document and vice versa is described in two XSL documents in the

XSL programming language, see section 2.5. The parser that does the actual conversion can be obtained from a variety of vendors and in nearly any programming language such as Java, PL/SQL, C, etc. Since all the parsers in fact are implementations of the XSLT interfaces that the W3C has defined [2], it is possible to performance test each one of them and then make a decision as to which one of them to use.
There are in fact competitors to the Corus/ALS© system, like the Microsoft BizTalk server, that uses XSL Transformations to convert all the way from one XML format to another. But since XSL Transformations are not optimal for string conversions and mathematical operations as discussed by Michael Kay in XSLT Programmer's Reference [16], most existing products (including the BizTalk server) use workarounds to achieve performance in these areas. The BizTalk server, for example, uses the Microsoft XML processor (MSXML) that can understand a XSL document mixed with Visual Basic code to handle what Microsoft thinks that the XSL language is not optimal for. This means that Microsoft has in fact bent the rules for the XSL Transformation language set up by the W3C and makes it impossible to use the XSL documents created by any other parser.
Since the Corus/ALS© system has all the functionality needed for string conversions, mathematical operation, etc. implemented as stored procedures in the database there is no need to do these operations in the XSL processor and they can thus be used in the most optimized way. Figure 3-1 shows the way the interpreter works.



1. **The interpreter captures the DTD through the reference in the XML document.**
2. **Using the DTD the metadata XML document is created and used to create the relational database model in the Corus database.**
3. **Using the DTD the interpreter is able to create two XSL documents. One for transforming a XML document to the internal XML format and one for transforming an internal XML document to the external format.**
4. **Now the XSL Parser can use the two XSL documents to do the transformations.**
5. **When a internal XML document has been created by the parser the data can be inserted into the database directly since the internal XML format refers to the correct relational database model.**

Figure 3-1 From XML to a DB via the interpreter

3.2 The Metadata XML format

When the interpreter analyses the DTD of a new XML document it needs a way of describing the relational database model that needs to be created. If there is a need to deploy the solution on some other platform there is a need for an extensible format that is not platform dependent. The decision therefore fell on a metadata XML format that could be easily exchanged between databases and platforms. As shown in Figure 3-1, the interpreter will create the metadata XML document that will be used for creating the relational database model. This metadata XML format must be able to create all the necessary tables, columns and keys that is part of the database model. The general idea was that this XML format could be used not only for the purpose of creating a relational database model for a XML format but also to tell the Corus/ALS© system the internal structure of the client systems it was connected to. Furthermore the metadata XML format could eventually be used for recreating the entire inner structure of the Corus/ALS© system, making it possible to move the implementation to another database, though this is outside the scope of this thesis. Figure 3-2 shows

the proposed DTD of the metadata XML format and Figure 3-3 shows a sample document that could be used for creating a relational database model.

```
<!ELEMENT DatabaseSchema (DatabaseTable*,Sequence*)>
<!ELEMENT DatabaseTable (Columns,Keys,Indexes)?>
<!ATTLIST DatabaseTable Name CDATA #REQUIRED>
<!ATTLIST DatabaseTable TableSchema CDATA #REQUIRED>
<!ELEMENT Columns (Column*)>
<!ELEMENT Column EMPTY>
<!ATTLIST Column Name CDATA #REQUIRED>
<!ATTLIST Column DataType CDATA #REQUIRED>
<!ATTLIST Column DataTypeName CDA TA #REQUIRED>
<!ATTLIST Column Size CDATA #REQUIRED>
<!ATTLIST Column DecimalDigits CDATA #REQUIRED>
<!ATTLIST Column Nullable CDATA #REQUIRED>
<!ELEMENT Keys (PrimaryKey?,ForeignKey*)>
<!ELEMENT PrimaryKey (PrimaryKeyColumn+)>
<!ATTLIST PrimaryKey Name CD ATA #REQUIRED>
<!ELEMENT PrimaryKeyColumn EMPTY>
<!ATTLIST PrimaryKeyColumn Name CDATA #REQUIRED>
<!ATTLIST PrimaryKeyColumn Order CDATA #REQUIRED>
<!ELEMENT ForeignKey (ForeignKeyColumn+)>
<!ATTLIST ForeignKey Name CDATA #REQUIRED>
<!ELEMENT ForeignKeyCol umn EMPTY>
<!ATTLIST ForeignKeyColumn Name CDATA #REQUIRED>
<!ATTLIST ForeignKeyColumn ReferencedSchema CDATA #REQUIRED>
<!ATTLIST ForeignKeyColumn ReferencedTable CDATA #REQUIRED>
<!ATTLIST ForeignKeyColumn ReferencedColumn CDATA #REQUIRED>
<!ATTLIST Fore ignKeyColumn Order CDATA #REQUIRED>
<!ELEMENT Indexes (Index*)>
<!ELEMENT Index (IndexColumn+)>
<!ATTLIST Index Name CDATA #REQUIRED>
<!ATTLIST Index Unique CDATA #REQUIRED>
<!ELEMENT IndexColumn EMPTY>
<!ATTLIST IndexColumn Name CDATA #REQUIRED>
<!ATTLIST  IndexColumn Sequence CDATA #REQUIRED>
<!ATTLIST IndexColumn Order CDATA #REQUIRED>
<!ELEMENT Sequence EMPTY>
<!ATTLIST Sequence Schema CDATA #REQUIRED>
<!ATTLIST Sequence Name CDATA #REQUIRED>
```

Figure 3-2 The DTD for the metadata XML format

```
<DatabaseSchema>
  <DatabaseTable Name="EMP" TableSchema="SYSTER">
    <Columns>
      <Column Name="EMPNO" DataType="3" DataTypeName="NUMBER" Size="4"
DecimalDigits="0" Nullable="NO" />
      <Column Name="ENAME" DataType="12" DataTypeName="VARCHAR2" Size="10"
DecimalDigits="0" Nullable="YES" />
      <Column Name="JOB" DataType="12" DataTypeName="VARCHAR2" Size="9"
DecimalDigits="0" Nullable="YES" />
      <Column Name="MGR" DataType="3" DataTypeName="NUMBER" Size="4"
DecimalDigits="0" Nullable="YES" />
      <Column Name="HIREDATE" DataType="93" DataTypeName="DATE" Size="7"
DecimalDigits="0" Nullable="YES" />
      <Column Name="SAL" DataType="3" DataTypeName="NUMBER" Size="7"
DecimalDigits="2" Nullable="YES" />
      <Column Name="COMM" DataType="3" DataTypeName="NUMBER" Size="7"
DecimalDigits="2" Nullable="YES" />
      <Column Name="DEPTNO" DataType="3" DataTypeName="NUMBER" Size="2"
DecimalDigits="0" Nullable="YES" />
    </Columns>
    <Keys>
      <PrimaryKey Name="PK_EMP">
        <PrimaryKeyColumn Name="EMPNO" Order="1" />
      </PrimaryKey>
      <ForeignKey Name="FK_DEPTNO">
        <ForeignKeyColumn Name="DEPTNO" ReferencedSchema="SYSTER"
ReferencedTable="DEPT" ReferencedColumn="DEPTNO" Order="1" />
      </ForeignKey>
    </Keys>
    <Indexes>
      <Index Name="PK_EMP" Unique="YES">
        <IndexColumn Name="EMPNO" Sequence="" Order="1" />
      </Index>
    </Indexes>
  </DatabaseTable>
  <Sequence Schema="SYSTER" Name="xm_EMP_seq" />
</DatabaseSchema>
```

Figure 3-3 An example of a metadata XML document

The metadata XML format is not intended to replace SQL but will be able to create the relational database models that is needed to take care of the XML documents that needs to be imported. The metadata format will not for example be able to create stored procedures and since different databases use different scripting languages for stored procedures it is not possible to describe them in a common way either.
For a person that is familiar with relational databases the metadata format should be quite strait forward. Each table is delimited by <DatabaseTable> tags and inside these tags there are three categories of tags:

<Columns>	used to group the columns together.
<Keys>	used to group the primary and foreign keys together.
<Indexes>	used to group the indexes together.

The <Columns> tag holds all the columns, which are defined by the attributes in each <Column> tag. The attributes are:

Name	The name of the column to create.
DataType	The Java SQL data type. This is the way that JDBC describes the data type of the column in a platform independent manner.
DataTypeName	The platform dependent data type name. Unfortunately not all of the jdbc drivers can understand the internal data types of a database properly, which makes it necessary to have this attribute.
Size	The size in bytes that the column should have in the database.
DecimalDigits	The number of decimal digits the column should possess. This only makes sense if the format is of the NUMBER type.
Nullable	Indicates if the column should be null able or not.

The <Keys> tag holds the <PrimaryKey> tag and zero or more <ForeignKey> tags. The name of the primary key can be found in the Name attribute of the <PrimaryKey> tag and the <PrimaryKey> tag in turn holds the <PrimaryKeyColumn> tag, which has a Name attribute that is the name of the primary key column and a order attribute that states the order of the columns of the primary key if there are more than one column. Likewise has the <ForeignKey> a Name attribute for the name of the foreign key and the <ForeignKey> tag holds the <ForeignKeyColumn> tag, which has the following attributes:

Name	The name of the column that has a foreign key constraint.
ReferencedSchema	The schema that the foreign column belongs to.
ReferencedTable	The table that the foreign column belongs to.
ReferencedColumn	The name of the foreign column.
Order	The order of the columns of the foreign key if there is more than one column in the foreign key.

The <Indexes> tag holds <Index> tags for all indexes belonging to the table. Each <Index> tag has a Name attribute for its name and a Unique attribute to indicate if the index is unique or not. Furthermore the <Index> tag holds the <IndexColumn> tag for the columns that are part of the index. The <IndexColumn> tag has a Name attribute for the name of the column and a Sequence attribute for indicating if the index should be sorted in any other way than the default way and a Order attribute for the order of the columns within the index.
It is also possible to create sequences with the help of a <Sequence> tag, which is a necessity as will be discussed later in section 3.3. The <Sequence> tag has a Name attribute for it’s name and a Schema attribute for the schema the sequence will be created for.

3.2.1 Choosing parser interface

There are essentially two different parser interfaces that are used today; the SAX interface [13] and the DOM interface [11]. They are both discussed in section 2.4 and in section 2.3. When using DOM the parser reads the entire XML document into memory and after that it is possible to manipulate the document. When using the SAX interface on the other hand, the parser reads the content of each element and when moving on to the next element it garbage collects the previous element, making SAX both fast and memory efficient. In this case however the data is limited and there might be a reason to show the user a graphical representation of the relational data model that will be created so the user can make changes before it is finally created. Thus the obvious choice is DOM since it builds an in memory representation of the XML document which can easily be mapped to a graphical user interface. As discussed in section 2.3 the DOM Level 1 implementation will do just fine for our interpreter since DOM Level 1 contains the necessary functionality for manipulating XML 1.0 documents.

3.2.2 Making an extensible implementation with DOM

It is also of great importance to make the implementation of the metadata XML format as extensible as possible if it turns out that it will be used later not only for the interpreter but also for describing more complex database structures as would be necessary if an entire Corus/ALS© repository containing stored procedures, views, and other more platform dependent structures would be described using the metadata format.
Though it would be possible to parse through the entire XML document with a single class, it would make it nearly impossible for someone else to read the code and understand what's going on. Two different approaches are discussed below, one for creating a relational database model from a metadata document and one for creating a metadata document from a relational database model.

3.2.3 Importing metadata

When creating a relational database model from the metadata document the DOM parser will read the entire document into memory. Instead of having all the logic in one giant class, the approach is instead to have one class for each type of element. Each class will then create one or several instances of the classes that will handle the sub elements of the current element and pass the sub elements along to the new classes. This will make the implementation very extensible and easy to follow since all that is needed to be able to handle new elements is the addition of new classes that handle the new elements and a few lines of code to instantiate and call the new classes. A JDBC call will then be executed for each table to create that table. See Appendix B: The complete code, for the code.

3.2.4 Exporting metadata

It is also possible to create a metadata document from an existing relational database model so that it can be exported to another database. As when importing data there is one class for each element that shall be created. The classes also inherit from an implementation of the org.w3c.dom.Element class called XMLElement that was made by Oracle. By extending the XMLElement class the classes themselves can be treated as XML elements. Thus will each class instantiate the subsequent classes and append them as children to itself. This should allow anyone who later wants to expand the metadata format to do this without having to change much of the old code. See Appendix B: The complete code, for the code.

3.3 The import/export XML format

As mentioned earlier, there was a need for an internal XML data format. After the relational database model has been created this internal XML format will be used to import and export data into and from the database model. The internal data format could also, in the future, be used to transfer data directly from one database to another. Since it could also be used for this purpose the overhead should be kept to a minimum.
Figure 3-4 shows the DTD for the internal XML format. Again this format is not intended to replace SQL, but is instead created solely for the purpose to be able to import or export data to and from the relational database models created by the metadata XML format. However, the thought is that it should be possible to extend this format in the future so that it could be used for other purposes as well.

```
<!ELEMENT data (transaction*)>
<!ELEMENT transaction (insert|update|delete)*>
<!ELEMENT insert (ref,row)>
<!ELEMENT update (ref,row,condition)>
<!ELEMENT delete (ref,row,condition)>
<!ELEMENT ref EMPTY>
<!ATTLIST ref schema CDATA #REQUIRED>
<!ATTLIST ref table CDATA #REQUIRED>
<!ATTLIST ref id ID #REQUIRED>
<!ATTLIST ref refid " -1">
<!ELEMENT row (d*)>
<!ELEMENT d #PCDATA>
<!ATTLIST d col CDATA #REQUIRED>
<!ELEMENT condition #PCDATA>
```

Figure 3-4 The DTD of the internal XML format

Figure 3-5 shows an example of a message conforming to the DTD in Figure 3-4.

```
<data>
 <transaction>
  <insert>
   <ref schema="SCOTT" table="TPA" id="ID8" refid="-1" />
   <row>
    <d col="TPAName">OBIStandard</d>
    <d col="Protocol">OBI</d>
   </row>
  </insert>
  <insert>
   <ref schema="SCOTT" table="Member" id="ID49" refid="ID8" />
   <row>
    <d col="MemberId">777777777777777</d>
    <d col="IdCodeType">ZZ</d>
    <d col="Partyname">LargeCo</d>
   </row>
  </insert>
  <insert>
   <ref schema="SCOTT" table="Address" id="ID60" refid="ID49" />
   <row>
    <d col="AddressType">location</d>
    <d col="City">SmallTown</d>
    <d col="State">NY</d>
    <d col="Zip">10000</d>
    <d col="Country">USA</d>
   </row>
  </insert>
 </transaction>
</data>
```

Figure 3-5 An example of an internal XML document

When storing a XML message into a relational database model only the <insert> tag will be used. The explanation for this is that the communication mechanism will be responsible for checking if there are new messages to fetch. If a new message exists it will be stored in its entirety into the database. Thus the <update> and <delete> tags will never be used when storing XML documents in the database. If a XML dialect is encountered in the future which like the internal XML format actually tells whether the information that is sent should be regarded as an insert, update or delete to the existing information an approach, which involves the use of the <update> and <delete> tags might be worth looking into but this is outside the scope of this thesis. Because the <update> and <delete> tags will not be used for storing XML messages I will not give an in depth discussion of how they are used in this thesis.
As can be seen in the DTD in Figure 3-4, the root element is the <data> element. The <data> element can contain zero or more <transaction> elements which in turn can contain zero or more <insert>, <update> and <delete> elements.
The <insert> element indicates that there is data to be inserted into a database table. The table to insert the data into is pointed out by the <ref> element, which is a child element of the <insert> element.
The <ref> element has four attributes: schema, table, id and refid. The schema and table attributes are self- explanatory, pointing out the database schema in which the database table exists and where the data should be inserted. However the id and refid attributes need a little more explanation.

The id attribute of the <ref> element is of the ID attribute type. This means that no other <ref> element in the entire XML document can have an id attribute with the same value. The purpose of this is to separate rows of the same table.
The refid attribute is then used to refer to rows of data in other tables so that relations between data in the XML document can be described. Thus the value of the refid attribute will either be the value of an existing id attribute or, according to the DTD, the value "-1" to indicate that there is no relation to data in an other table.
A XML document will normally be broken into several database tables, as discussed in section 3.4, and the different parts of the original XML document are kept together by the id and refid attributes. Even though the id and idref attributes resembles the use of primary and foreign keys in a database, their values only have a meaning in the context of the XML document. Thus, the interpreter will need to handle the representation of the relations in the database when importing and exporting data as described in section 3.3.3 and in section 3.3.4.
Each <insert> element also contains a <row> element, which in turn contains zero or more <d> elements. It's the <d> elements that contain the data to be inserted into the database table. A <d> element has a 'col' attribute that refers to the column the data should be put in.

3.3.1 Choosing parser interface

As mentioned earlier there are essentially two different parser interfaces that are used today: the SAX interface and the DOM interface. They were both discussed in section 2.4 and in section 2.3. When using DOM, the parser reads the entire XML document into memory and after that it is possible to manipulate the document. When using the SAX interface on the other hand, the parser reads the content of each element and when moving on to the next element it garbage collects the previous element, making SAX both fast and memory efficient. In this case, where the data size can be almost unlimited and there is no user manipulation required, SAX is the ideal choice. It doesn't matter that SAX v1.0 has no way of parsing the DTD since the DTD is already well known because the documents that we will parse are of our own internal XML format.

3.3.2 Making an extensible implementation with SAX

It is even more important to think through an implementation made with SAX than one made with DOM. The reason for this is that SAX uses just a few events to indicate the beginning of an element, the end of an element, the attributes of an element and the data of an element. For a more thorough explanation of SAX see section 2.4. The class that implements the SAX event handler interface will thus be responsible for taking care of the processing of all elements that are encountered and the event handling methods of that class will do a number of unrelated tasks.
An approach to solve this problem using a design pattern is presented in Professional XML [15]. The two methods presented are a filter pattern and a rule-based pattern.
A filter design pattern is constructed by creating a number of classes that will each handle specific parts of the document to be parsed. The classes are then connected together by letting all the classes both handle events and passing the events to the next

class in the chain of classes. This pattern is sometimes called a pipeline pattern because the sections of a pipeline can resemble the stages of processing.
A rule-based pattern on the other hand uses a switch that looks at the events and then makes a decision of which actions should be taken. In this case the switch is a class that makes decisions based on the element types and then invokes the right event handling class for that element.
Both of these design patterns result in fast processing and easily maintainable code but if a new element type is introduced the rule-based design pattern is preferable since the only addition to the old code is to register a new element handler in the switch class.

3.3.3 Importing data

When taking data from the internal XML format and storing it in the relational model the decision fell on the rule-based design pattern because of the possibility that the internal XML format will be extended in the future.
First the XML document is parsed quickly and the id and refid attributes are examined to determine the relations between the data to be inserted. After the relations are determined the document is parsed again and the data is inserted into the tables.
In the relational database model each table will contain a column called ID that is the primary key to uniquely identify the rows. The value of the ID column is taken from a database sequence when each row is created. There will be one database sequence for every ID column. Furthermore, there will be a IDREF column to relate rows of different tables with each other. Every IDREF value will thus be equal to one ID value in the related table. The IDREF column is however absent in the topmost table since the rows of this table will not have a relation to another table.
During the first parsing through the XML document the values of the ID and IDREF columns is determined by getting the next value from the database sequences so when the document is parsed through again all the rows can be created with the right ID and IDREF values. For the sake of clarity the process is also shown in Figure 3-6.
The values of the id and refid attributes are used for keeping the relation between the data in the XML document and should not be confused with the values of the ID and IDREF columns. Since the values of the id and refid attributes are only unique within one single XML document, they cannot be used when inserting the data into the database tables.

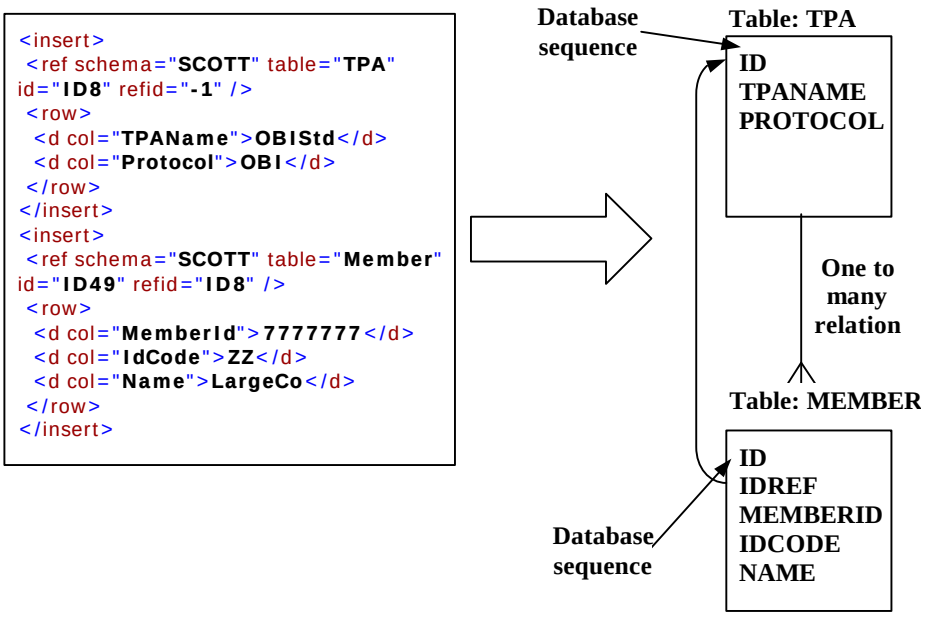


Figure 3-6 The relation between id and refid attributes and the ID and IDREF columns

The code can be found in Appendix B: The complete code.

3.3.4 Exporting data

When recreating a XML document, the first step will be to create an internal XML document and then use XSLT to convert it to the final XML format, see Figure 3-1. The internal XML document will be recreated from the relational database model for that XML format. Each relational database model will have a top- table that is used for storing the root element and possibly some more element of the XML document, see section 3.4. When recreating the internal XML document from the relational database model, the correct row of the top- table must first be found. Since the interpreter knows the relations between the tables in the relational database model, it will be able to get the right rows in the other tables by comparing the value of the IDREF column in each of these tables to the value of the ID column of the table it has a relation to, as can be seen in Figure 3-6.
The relations between the data must also be maintained in the internal XML format. This can be achieved by letting the interpreter create an internal index for the id attributes. If the interpreter keeps track of the id attributes used so far it will also be able to create refid attributes that refer to the id attributes in the same way that relations are maintained in the database.

3.4 Finding the structure of a foreign XML dialect

Before the XSL documents that will do the transformation to the internal XML format can be created, the DTD of the external XML format needs to be examined. The DTD describes all combinations of messages that can possibly be created for that XML dialect. The DTD is therefore necessary to be able to create the XSL documents and a relational database structure that can take care of all possible messages. Since a XML document doesn't need to have a DTD, the interpreter should be fed a DTD when setting up the environment but when the environment has been created, the DTD should not be needed any more.
The W3C has defined an interface for DOM Level 1 to make it possible to parse the underlying DTD as well as the XML document itself. Using this interface, the interpreter should be able to build an internal representation of the structure of the XML dialect.

3.4.1 Using the DTD to find the structure

When using the interfaces from W3C to parse the DTD it is possible to get an understanding of the following:

- ?? What elements are possible as child elements of other elements?
- ?? Must an element appear at all? Will it appear zero or one time (?), one or more times (+) or zero or more times (*)?
- ?? What attribute type does the attributes have?
- ?? Does the attributes have any default values?

Using this information the interpreter will be able to create the relational database model to handle all possible messages for this XML dialect.

3.4.2 Mapping a XML dialect to a database structure

A XML document can only have one root element according to the W3C XML 1.0 specification [1]. This element is commonly known as the documents element. The document element in turn can have zero or more child elements. All the elements in the XML document except the document element can be declared to appear zero or more times, one or more times etc. as described above in section 3.4.1. This kind of structure resembles the structure in a database where a table can have a one to many relation to other tables. The proposed solution for storing the XML document is therefore to create new tables for each element and its children whenever an element declaration that states that the element can occur more than once is found. This solution also means that all the XML documents will be stored in a normalized way in the database. The names of the tables and of the columns can be generated automatically by the interpreter or chosen by a user through a GUI (graphical user interface) since the XSL processor will use the XSL documents to map to the right tables and columns, as described in section 3.5.
The XSL documents also describe how to rebuild the relation between the elements in the XML document, i.e. one element is the child of an other element, from the internal

XML format. Thus, there is no reason to maintain the relations between the original elements of the XML document in the relational database mode.
The result is a number of tables, as many as there are elements that can appear more than ones, with columns that will contain the attribute content and the element content.
Since data types are not part of the XML 1.0 standard, there is no way the interpreter can know how large an element actually is and what the ideal format in the database should be. This could however be implemented by letting the user set it himself in a GUI. The interpreter can however use a default size and type in the database though there is now way of knowing if it will work or not. The default value was chosen to a string of 200 bytes after having tested approximately 30 XML documents from 10 different XML dialects. However, by setting the type and value manually by a user, the size and speed of the database model can be greatly improved. Though this might sound like a great disadvantage it is not the case because almost every standard protocol that uses XML 1.0 states the type and size of the data that will be sent in their specifications.
The recommendation is not to use the standard size and type that the interpreter uses but to specify the types and sizes for every attribute and element to get maximum performance.
When the XML Schema specification becomes a recommendation and the interpreter is adapted to handle this specification as well, the user will not have to specify any data types and size because this is be done in the XSD, the equivalent of a DTD for XML Schema. So the interpreter will be even more automatic when used with XML Schema documents in the future. The operation of the interpreter will however not change much, so the adaptation of the interpreter to handle this should be fairly easy.

3.5 Transforming foreign XML dialects

When the relational database model has been created, the creation of the two XSL documents remain. These two documents will be used by the XSL processor (also called the XSL parser) to transform an external XML document to the internal XML format and vice versa, as can be seen in Figure 3-1.
The names of the tables and columns are explicitly defined in the XSL documents by the interpreter during the creation of the relational database model and the creation of the XSL documents. This ensures that the translation between the two formats is a matter of copying data from one element to another and it will make the process very fast.

3.5.1 Namespaces in external XML documents

Nearly a year after XML 1.0 became a W3C recommendation a standard called Namespaces in XML became a W3C recommendation. A XML namespace is used to organize names into distinct sets and avoid name collisions. With the use of XML namespaces it is possible to have elements and attributes with the same name but with different properties and usages.
Namespaces are however not meaningful in the relational database model since name collisions are avoided by storing data into different tables using different column

names. The namespaces can therefore be removed in the process of creating the internal XML document by the XSL processor and recreated by the XSL processor when creating the external XML document.

3.5.2 Choosing a design pattern

XSL Transformations is a functional programming language with origins from LISP languages. This means that XSLT is pure recursive language and this imposes some restrictions on the possible design patterns that can be used when creating a XSLT document. As stated in XSLT Programmer's Reference [16] most XSLT documents fall into four design patterns:

- Fill-in-the-blanks style sheets.
- Navigational style sheets.
- Rule-based style sheets.
- Computational style sheets.

There are of course also XSLT documents that fall into more than one of these design patterns at the same time.
A fill-in-the-blanks style sheet is an output oriented style sheet where the output tags are simply written by the author of the document and variable data is inserted by the addition of extra XSL tags inside the output tags. All tags are created at the encounter of the document element and data from other elements than the document element is found using a absolute path from the root of the document down to that element.
A navigational style sheet is also essentially an output oriented style sheet but with variables, subroutines and possibly constructs as keys, parameters and sorting. Thus it looks like a conventional procedural program, though variables in XSLT are a bit awkward because they cannot be changed after they have been instantiated.
A rule-based style sheet on the other hand is not output oriented at all. It uses XSL constructs to catch events from the XSL processor so that each element can be handled differently. A rule-based style sheet will therefore make minimal assumption about the structure of either the input document or the output document it will just make a decision of what to output when encountering a certain input element.
The computational style sheet is somewhat more complex that the other three design patterns. It is used when elements in the input document doesn't correspond directly to elements in the output document. For example, when the content of an element needs to be split up between several elements in the output document or when the contents of several elements in the input document needs to be concatenated and inserted as the content of one element in the output document. As mentioned earlier XSLT is a functional programming language and as a functional programming language it doesn't have assignment statements or variables, instead functions and the output from those are the only way of manipulating data.
The transformation that the XSL processor has to do between the external XML format and the internal XML format is a matter of mapping input elements content and attributes to output element content and attributes. Thus there is no need for a computational style sheet. The complexity of the external XML format can however be extensive so keeping track of relative paths to underlying elements, as with the

output oriented style sheets, is not very attractive. The decision falls therefore on the rule-based style sheet.

3.5.3 Relations between data in the internal XML format

The internal XML format is likely to be very different to the external XML format because it retains the relations between the data with the use of id and refid attributes as described in section 3.3, while the external XML format has a natural relation between elements due to the fact that elements have child elements. The hierarchical structure of the two documents is therefore also likely to be different.
The relations between the data in the internal XML format are very easy to obtain when exporting data because the interpreter itself will create the internal XML document and thus be responsible for maintaining the relations. However when importing documents, the XSL processor must be able to maintain the relations between the data because the XSL processor is responsible for creating the internal XML documents. Luckily the XSLT Version 1.0 standard from W3C [2] states that all nodes of the input document must have a unique id and that the id of every node must be obtainable through the ‘generate-id()’ XSL function. The id and refid attributes in the output document can then be obtained by using the ‘generate-id()’ function directly in the process of creating the output document. In the relational database model new tables are created for each element that can occur more than once so the id attributes will naturally be set to the id of those elements. The XSL processor can then find the refid attributes by referring to the elements that the id attributes was obtained from. The process of obtaining the id and refid attributes is shown in Figure 3-7.

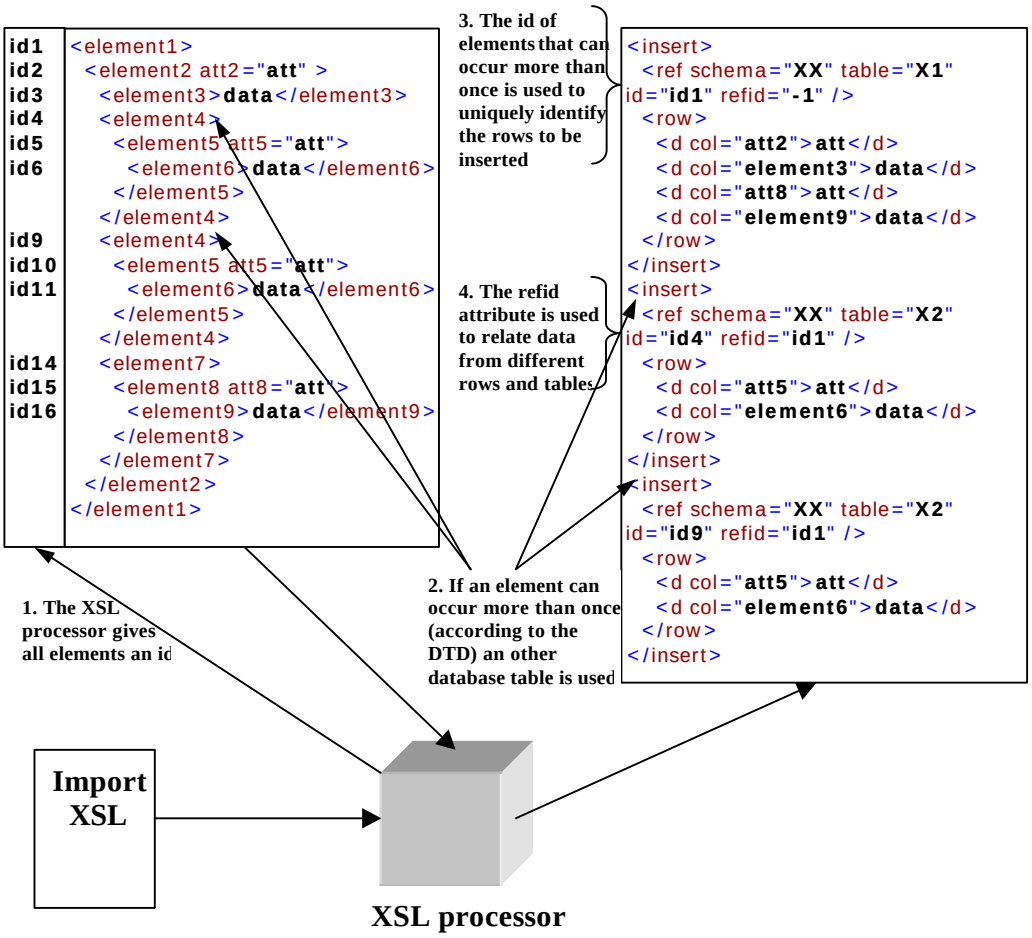


Figure 3-7 The transformation from the external XML document to the internal XML forma

3.5.4 The import XSL document

When the XSL processor creates the internal XML document it will create new <insert> elements for each row to insert into a table in the database. Each row to be inserted into a table is constructed from element content and attributes in the external XML document. When the interpreter examined the DTD of the external XML format it created a database model where new tables were created when the DTD indicated that an element could occur more than once. Thus the XSL processor will need to create new <insert> elements each time it encounters an element that can occur more than once. Only when a <insert> element and its child elements are created can the XSL processor continue to process the XSL constructs for the child elements that will create the next <insert> element.

3.5.4.1 The Style Sheet

The proposed style sheet that is created by the interpreter and used by the XSL processor is a combination of a rule- based style sheet and a navigational style sheet.

Since every <insert> element and its child elements needs to be created before moving on they all have to be created when the XSL processor encounters the first element of a new row. The XSL processor must therefore refer to the other elements that belong to the same row relatively to the position of the element it is currently processing. This is the reason that the style sheet cannot be a pure rule- based style sheet.
When the XSL processor is finished with creating the <insert> element and its child elements, the XSL constructs that belongs to any new elements that are the first ones of a new row will be called and so on.
The XSL processor will need to handle elements of the same type differently depending on the context of these elements in the original XML document. The reason for this is that the same type of element can be encountered in different parts of the same document and the <insert> elements that are created can refer to different database tables. The XSL constructs that the XSL processor uses when it encounters a new element are called templates and by giving the templates a mode attribute it is possible to decide which of the templates that should be used depending on the template that was used last. Using mode attributes is one way of deciding exactly in what way a XML document should be processed as explained further in XSLT Programmer's Reference [16]. Another popular way is to use parameters to pass variables between the templates and then use <xsl:if>- statements to process the variables and decide what to do. The decision fell on mode attributes because it will result in faster processing because the XSL processor will not have to make any decisions at all to decide what to output. All it has to do is to follow the path that is outlined by using the mode attributes. The downside of choosing mode attributes is that the XSL document could be more difficult to create by the interpreter.
Figure 3-8 shows an example of a part of a XSL document that passes element ids between templates to maintain the relation between the data, templates with mode attributes and relative paths to get the contents of elements and attributes.

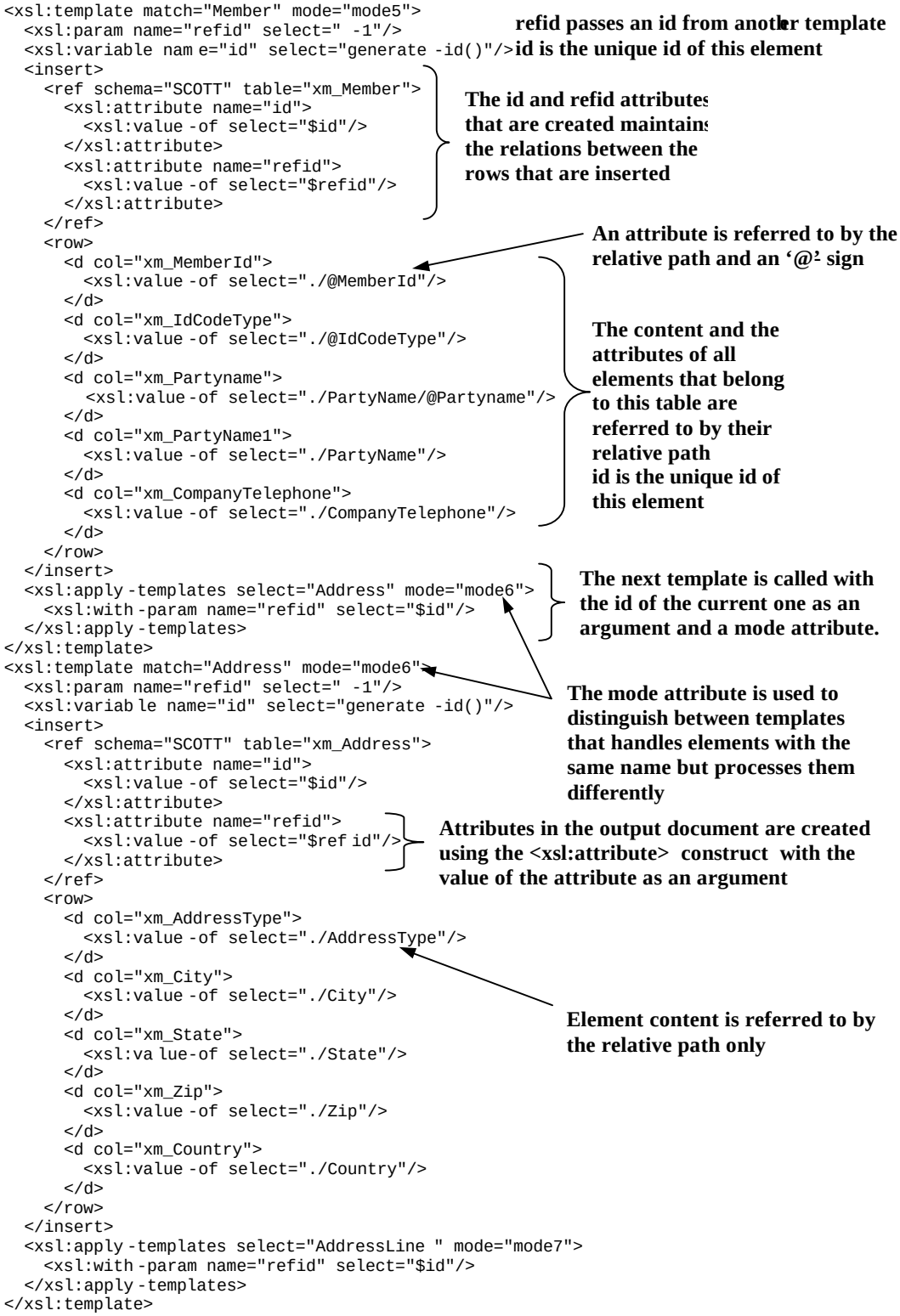


Figure 3-8 An example of an import XSL document

3.5.4.2 Creating the Style Sheet

The interpreter is responsible for creating the import and export XSL documents when it is examining the DTD of a new XML dialect. When the XSL documents are created, any XSL processor will be able use the XSL documents to do the necessary transformations, no matter if the XSL processor is implemented in C, Java or PL/SQL. This is one of the strengths of using XSL to transform the documents.
To be able to create the XSL documents the interpreter will have to build an internal model for both the export XSL document and the import XSL document, since the XSL documents are completely different. The models must be able to answer the following questions:

- ?? Should each attribute of an element map to an attribute or should it map to an elements content of the output document?
- ?? Should each elements content map to an attribute or to element content in the output document?
- ?? What elements and attributes will be a part of the same row of a database table?
- ?? How do the database rows relate to each other?
- ?? Where in the XSL document, or at the encounter of what element, should the output elements be created?

Before the XSL models are created, a model for the relational XML document is created. This model has all the information needed about the database tables that are going to be used and can thus be used later when the two XSL models are created.

3.5.5 The export XSL document

When the XSL processor creates the external XML document it will look for a <transaction> element and then start to extract data from the underlying <insert> elements and copy that data into the new output elements.
Each <insert> element represents a row of a table in the database. The relations between the <insert> elements are describing the natural relations that exist in the external XML format where elements are children of other elements as described in section 3.5.3. The XSL processor can recreate the hierarchical order of the elements simply by following all paths of correlated id and refid attributes from the document element and down.
All the output elements will have to be created by the XSL processor on the encounter of the <transaction> element since data from different <insert> elements can map to sub elements of other elements. Furthermore it is essential that an element has not been closed if a sub element is to be created.

3.5.5.1 The Stylesheet

The proposed style sheet that is created by the interpreter and used by the XSL processor is a navigational style sheet. The reason for this is that the hierarchy of the internal XML document is always known and all the output elements are created at the encounter of the <transaction> element. The style sheet will however be quite

complex to be able to recreate the hierarchical external XML document from the more flat internal XML document.
First the <insert> element that contains the data to recreate the document element and the topmost elements of the output document is found and the output elements are created. Then the id attribute of this <insert> element is used to find an <insert> element with a refid attribute that matches the id attribute. This newly found <insert> element is then used to create sub elements to the appropriate output element that was created from the former <insert> element. This goes on until all output elements has been recreated. An example of a style sheet that could be created by the XSL processor can be found in Figure 3-9. The output from the XSL processor using this style sheet would resemble the XML document in Figure 3-10, where xxx means data from the relational database model.

Figure 3-9 An example of an export XSL document

```
<TPA>
  <TPAInfo>
    <TPAName>xxx</TPAName>
    <TPAType>
      <Protocol>xxx</Protocol>
      <Version>xxx</Version>
      <Type>xxx</Type>
    </TPAType>
    <Role>
      <RoleDefn>
        <RoleName>xxx</RoleName>
        <RolePlayer>xxx</RolePlayer>
      </RoleDefn>
    </Role>
    <Participants>
      <Member MemberId="xxx" IdCodeType="xxx">
        <PartyName Partyname="xxx">xxx</PartyName>
        <CompanyTelephone>xxx</CompanyTelephone>
      </Member>
    </Participants>
  </TPAInfo>
</TPA>
```

Figure 3-10 The output from a transformation done with the previous style sheet

3.5.5.2 Creating the Stylesheet

As stated in section 3.5.4.2, the interpreter will create an internal model for the two XSL documents. These models are then used to create the documents.
Before the XSL models are created a model for the relational XML document is created. This model has all the information needed about the database tables that are going to be used and can thus be used later when the two XSL models are created. If a graphical user interface is created it can be used to manipulate the names and types of the tables and columns for the relational model and this will reflect in the two XSL models as well.

3.6 XML documents with cyclic redundancy

A XML document is said to have a cyclic redundancy if its DTD states that an element of a certain type can have children or grand children of the same element type. This implies that a XML document conforming to the DTD can have an infinite depth. Documents with cyclic redundancy thus have to be treated differently than XML documents without cyclic redundancy.
There are a number of ways to store a document with cyclic redundancy in the database, some of them being more or less naïve. Three methods are discussed here: treating cyclic elements as element content, introducing a finite depth and cyclic database model design.

3.6.1 Cyclic elements as element content

The simplest way of storing XML documents that conform to a DTD with cyclic redundancy is to create a database model that stores all cyclic elements and their sub elements in a single column with all the tags intact. This means that the documents will be treated as if they weren't cyclic at all.

The disadvantage of this method is that there is no simple way of extracting data from elements that are cyclic. This method should thus only be used when there is no need to use the data that appears in the cyclic elements and is therefore of limited use. An example of a document with cyclic data that might not be interesting is a document that stores a complete document of the same type as one of its sub elements. The sub document might for example be used to follow a conversation and the interest lays in the new data, not the old.

3.6.2 Introducing a finite depth

A DTD can be used to describe cyclic redundancy as described earlier but there is no way to limit the depth of reoccurring elements. Thus a DTD cannot be used to describe a document format where a certain element can contain a sub element of the same type but that sub element will not contain another sub element of the same type. A creator of a XML dialect that will be used in this way can however describe this in the specification of the format. When one knows that the depth is not unlimited it is possible to treat the XML documents in the same way as a document that is not cyclic at all. This could then be set in a graphical user interface.

3.6.3 Cyclic database model design

The most sophisticated approach would be to create a database model that can hold cyclic XML document. A way of doing this is to have three database tables, a table for document information, a table for element information and references and a table for attributes. A simple example of this database design is shown in Figure 3-11.

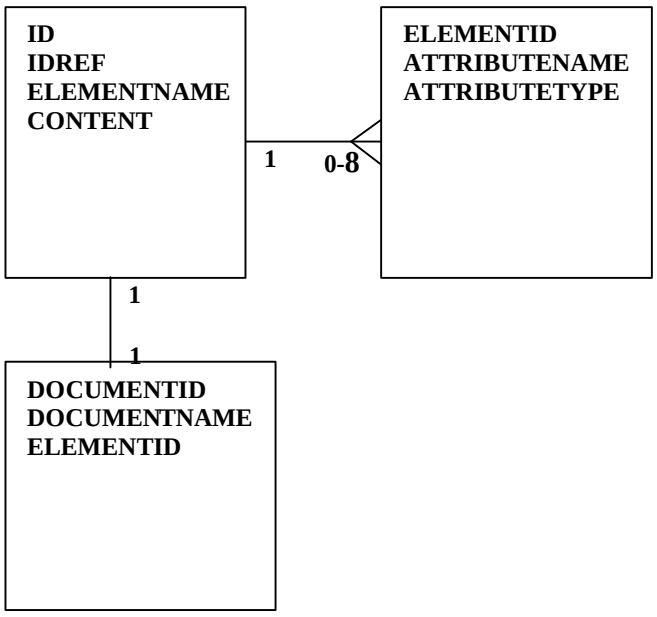


Figure 3-11 A cyclic database model design

Even if both cyclic and non-cyclic documents could be stored in a cyclic database model it poses problems. The problems arise when PL/SQL is going to be used to translate between different database models. Stored procedures, database triggers etc. will have to be a lot more sophisticated and if there is a need to translate to a database model that resembles a client system database model it will surely not be built in the same way.

3.6.4 Choosing a method for cyclic XML dialects

Even though it is tempting to store all XML documents in a cyclic database model, it poses a lot of problems that will not exist if a non-cyclic database model is used. After having examined the 122 different XML dialects that the Open Applications Group has posted on its website it, turned out that none of these XML dialects were cyclic. Open Applications Group is one of the biggest organizations for creating XML dialects in the world and has currently 66 members including IBM, Oracle, SAP Microsoft, etc. This means that there is a relevant reason to treat the few cyclic XML dialect that exists in a separate way.
Also when using the Corus/ALS© system to connect client systems a model of the client system is built inside the Corus/ALS© database. The database models that resemble the connected client systems are almost inevitably structured in an ordinary relational structure and the simplest way to integrate between the formats is when they resemble each other.
Even if a cyclic XML dialect is encountered, the interpreter can still handle it by introducing a finite depth as described in section 3.6.2. It is only in cases where it is impossible to choose a depth that will never be exceeded that the cyclic database model is needed. By using different database models for client systems that use the same XML format the possibility that a depth can be found increases even more. Microsoft, one of the industry leaders, has introduced a depth attribute for cyclic XML dialects in its BizTalk server to get rid of the finite cyclic redundancy problems. The maximum depth in the BizTalk framework is set to seven cyclic occurrences of the same element type. There are currently around 500 XML dialects that have been adapted to work with the BizTalk framework.

3.7 Putting it together

The construction of the interpreter is the first step to integrate this functionality into the Corus/ALS© product. Before the interpreter can be used with the Corus/ALS© product, the database models that the interpreter creates must be created in the metadata format that the Corus/ALS© system uses in its repository. The repository will then be used to maintain and created the database models as with all other integrations that are made with the Corus/ALS© system. There must also be a way for the Corus/ALS© system to maintain and use the XSL files that are created by the interpreter and the best way to do this is probably to store the XSL documents in the repository and load them directly from a table.

The Corus/ALS© system will also be responsible for the transport mechanisms involved and for feeding documents through the XSL processor with the appropriate XSL documents.
When the Corus/ALS© system is adapted to use the interpreter integrations to and from different XML dialects will be possible and the powerful functionality in the Corus/ALS© system to transform, log and migrate data in a transactional way can be fully utilized.

4 Evaluation

The purpose of the thesis was to determine if it was possible to find a way to examine a XML document and its DTD and create a relational database model to hold documents of that type. After having created the relational database model there should be a way of quickly store and retrieve documents from the relational database model. The solution must be easily extensible and the environment should be set up from a repository or a XML document.
The proposed solution accomplishes all the requirements and relies heavily on the standards outlined by W3C making it platform and vendor independent. This section gives a brief discussion of the design issues and the choices that has been made to be able to accomplish this.
First three different approaches to build the interpreter were discussed.
The first approach discussed was to let the interpreter compare all XML documents it received to their DTD's and use the DTD's to make decisions on what relational database model to use. The second approach was to create configuration files that the interpreter could use to make a decision of what relational database model to use. The last approach that was discussed was to use XSLT to transform all documents to an internal format that directly described what relational database model to use and how to put the data into that model.
The decision fell on the approach where XSLT was used to transform all documents into an internal XML format because it would make the interpreter very small, fast and maintainable. No decisions need to be made by the interpreter during operation because the connections to the database are hard coded into the XSL documents that are used by the XSL processor during the transformation process from the external XML document to the internal XML format and vice versa. The XSL processor can also be replaced at anytime if a faster XSL processor is encountered because all the XSL processors build on the W3C XSLT recommendation.
When setting up the environment the W3C Document Object Model (DOM) was used to create models for the transformation documents and the relational database model. This means that a graphical user interface can be constructed at a later point to let a user decide the schema, table names, column names, and data types for the relational database model.
When parsing the output from the XSL processor the Simple API for XML (SAX) was used because it is faster and more memory efficient, than the Document Object Model (DOM). A rule based design pattern was used to make the actual implementation fast and easily maintainable.

It is in the XSL transformation where most of the work is done and it is essential that they are as efficient as possible. For this reason a lot of emphasis were put on the design patterns used in the XSL documents. The most effective design pattern for the import XSL document turned out to be a combination of a rule based style sheet and a navigational style sheet. The most effective design pattern for the export XSL document turned out to be a navigational style sheet because the internal XML documents have a determined hierarchy.
Both the import and export XSL transformations are of a pure “copy and paste” nature since all string and mathematical operations will be conducted in the relational database using stored procedures. This is another reason why the transformations will be extremely fast, because this is the kind of transformations that the XSLT language was created for.
Cyclic XML dialects will have to be treated differently than ordinary XML dialects because there is no way of knowing how deep the hierarchy of a cyclic XML dialect will extend. The proposed solution is to let the user decide how deep a cyclic element will extend. This is the same way that Microsoft’s BizTalk server handles cyclic XML dialects.

5 Conclusion

An interpreter that works along the lines described earlier was implemented. The implementation shows that it is quite feasible to construct a configurable interpreter that can create a relational database model and both store and retrieve XML documents from that relational database model. The interpreter does not take XML namespaces into account because it is out of scope for this thesis and a fairly straightforward matter of implementing that can be left as something to do before making a commercial version of the interpreter.
The implementation shows the strength of the W3C XSL transformations language and is believed to be both fast and easily maintainable due to an in depth look at the use of DOM, SAX and different design patterns.
There are however a few details that would have to be sorted out before product deployment would be possible.
The intention is that the interpreter is going to be a part of the Corus/ALS© system. The Corus/ALS© system uses a repository where all transformations and internal relational database models are described in metadata format. If the interpreter is going to be used by the Corus/ALS© system it must first be manageable from the same graphical user interface as the Corus/ALS© system uses. Secondly all configuration documents that the interpreter produces must be maintained through the Corus/ALS© repository and the relational database models must not be created directly from XML documents but instead from the repository. Last but not least the communication mechanisms of the Corus/ALS© system needs to be adapted to be able to poll for new XML documents and feed them to the interpreter.

6 Future work

Before the interpreter can be used it has to be adjusted to work with the Corus/ALS© graphical user interface and it's repository. Furthermore should the interpreter take XML namespaces into account so that the two XSL transformations removes and recreates the namespaces that might be used in the external XML documents.
When the XML Schema becomes a W3C recommendation the interpreter will have to be extended to be able to interpret XML Schema documents as well and this will mean that the interpreter will become even more automated since the XML Schema describes the data types that are used, which is not possible with XML 1.0.

References

[1] Tim Bray, Jean Paoli and C. M. Sperberg-McQueen. 1998. Extensible Markup Language (XML) 1.0. REC-xml-19980210. Online. W3C. Available: http://www.w3.org/TR/REC-xml 20 September 2000.

[2] James Clark. 1999. XSL Transformations (XSLT) Version 1.0. REC-xslt-19991116. Online. W3C. Available: http://www.w3.org/TR/xslt 20 September 2000.

[3] Tim Bray, Dave Hollander, Andrew Layman. 1999. Namespaces in XML. REC-xml-names-19990114. Online. W3C. Available: http://www.w3.org/TR/REC-xml-names 20 September 2000.

[4] James Clark and Steve DeRose. 1999. XML Path Language (Xpath) Version 1.0. REC-xpath-19991116. Online. W3C. Available: http://www.w3.org/TR/xpath 20 September 2000.

[5] David C. Fallside. 2000. XML Schema Part 0: Primer. WD-xmlschema-0-20000407. Online. W3C. Available: http://www.w3.org/TR/xmlschema-0 20 September 2000.

[6] Henry S Thompson, David Beech, Murray Maloney and Noah Mendelsohn. 2000. XML Schema Part 1: Structures. WD-xmlschema-1-20000407. Online. W3C. Available: http://www.w3.org/TR/xmlschema-1 20 September 2000.

[7] Paul V. Biron and Ashok Malhotra. 2000. XML Schema Part 2: Datatypes. WD-xmlschema-2-20000407. Online. W3C. Available: http://www.w3.org/TR/xmlschema-2 20 September 2000.

[8] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman and Steve Zilles. 2000. Extensible Stylesheet Language (XSL) Version 1.0. WD-xsl-20000327. Online. W3C. Available: http://www.w3.org/TR/xsl 20 September 2000.

[9] James Clark. 2000. Associating Style Sheets with XML documents Version 1.0. REC-xml-stylesheet-19990629. Online. W3C. Available: http://www.w3.org/TR/xml-stylesheet 20 September 2000.

[10] Bert Bos, Håkon Wium Lie, Chris Lilley and Ian Jacobs. 1998. Cascading Style Sheets, Level 2 CSS2 Specification. REC-CSS2-19980512. Online. W3C. Available: http://www.w3.org/TR/REC-CSS2 20 September 2000.

[11] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson and Lauren Wood. 1998. Document Object Model (DOM) Level 1 Specification. REC-DOM-Level-1-19981001. Online. W3C. Available: http://www.w3.org/TR/REC-DOM-Level-1 20 September 2000.

[12] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Laurence Cable, Mike Champion, Mark Davis, Joe Kesselman, Philippe Le Hégaret, Tom Pixley, Jonathan Robie, Peter Sharpe and Chris Wilson. 2000. Document Object Model (DOM) Level 2 Specification. CR-DOM-Level-2-20000510. Online. W3C. Available: http://www.w3.org/TR/DOM-Level-2 20 September 2000.

[13] David Megginson. 1998. SAX 1.0: The Simple API for XML. Online. Megginson Technologies. Available: http://www.megginson.com/SAX/SAX1/index.html 20 September 2000.

[14] David Megginson. 2000. SAX 2.0 The Simple API for XML. Online. Megginson Technologies. Available: http://www.megginson.com/SAX/index.html 20 September 2000.

[15] Didier Martin, Mark Birbeck, Michael Kay, Brian Loesgen, Jon Pinnock, Steven Livingston, Peter Stark, Kevin Williams, Richard Anderson, Stephen Mohr, David Baliles, Bruce Peat and Nikola Ozu. 2000. *Professional XML*. ISBN 1-861003-11-0, Birmingham UK: Wrox Press Ltd.

[16] Michael Kay. 2000. *XSLT Programmer's Reference*. ISBN 1-861003-12-9, Birmingham UK: Wrox Press Ltd.

[17] Bob DuCharme. 1999. XML The Annotated Specification. ISBN 0-13-082676-6, NJ USA: Prentice Hall PTR.

[18] Michael Leventhal, David Lewis and Matthew Fuchs. 1998. Designing XML Internet Applications. ISBN 0-13-616822-1, NJ USA: Prentice Hall PTR.

Appendix A: Acronyms and Abbreviations

ALS	Application Linking System
EAI	Enterprise Application Integration
XML	Extensible Markup Language
EDI	Electronic Data Interchange
RDBMS	Relational Database Management System
W3C	World Wide Web Consortium
SGML	Standard General Markup Language
DTD	Document Type Definition
EBNF	Extended Backus Naur Form
DOM	Document Object Model
SAX	Simple API for Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformation
XPointer	XML Pointer language
Xpath	XML Path Language
HTML	Hypertext Markup Language
CSS	Cascading Style Sheet
URI	Uniform Resource Identifier
URL	Uniform Resource Locators
HTTP	Hypertext Transfer Protocol
SQL	Structured Query Language
JDBC	Java Database Connectivity

Appendix B: The complete code

This section has been left out completely and is for Corus internal use only.