XML Schema Support in MXQuery

Kostis Tsoulos
Master’s Thesis

Databases and Information Systems Group

2008

Supervision: Dr. Peter Fischer
Prof. Dr. Donald Kossmann
Abstract

XQuery, a powerful query language for querying XML data, and XML Schema, a language for expressing a schema—a set of rules describing the structure and data content of XML documents—are established XML technologies and W3C standards.

The W3C Recommendation for XQuery describes the option of using and importing XML schemas in XQuery queries, allowing for the assignment of types to data and validation of both input XML and query results.

The feature of schema-awareness provides the XQuery processor with information about definitions included in schemas, resulting in better optimized and tested queries. This information can be exploited to facilitate and support predictability, early detection of errors, optimizations, special processing based on type and the validity of query results.

At the same time, data stream processing has become a hot topic. One of the issues that the research community tries to resolve is the fact that streaming is dealing—by its very nature—with infinite data sources and in most cases a streamed element can be seen and accessed only once.

A recent research proposal provides a formal specification of schema for describing the properties of streams. The availability of such stream schema information can enable optimizations and early detection of problems arising with infinite data. For the special case of XML streams, the proposal also describes an XML schema extension.

The main goal of this Master Thesis is the integration of XML Schema-awareness in the MXQuery engine and the exploration of implications that the availability of such a feature has for query processing, streaming execution and optimizations. For the needs of streams processing, the implementation of the XML Schema extension for streams and the support of a tool for XML Stream parsing and validation constitute further goals of this thesis. MXQuery is a Java-based XQuery and XQueryP engine that uses a low-memory footprint and supports streaming execution.
Acknowledgments

I would like to thank Professor Dr. Donald Kossmann for the assignment of the subject of the thesis and his valuable advice as a mentor throughout my studies. Special thanks to Dr. Peter Fischer for devoting precious time to the analysis and resolution of the issues that came up during the preparation of this thesis and his continuous support during my studies at the ETH. Our discussions on technical and research issues have been extremely interesting and influential. Finally, I would also like to thank Kyumars Sheykh Esmaili for his active participation in the discussions that took place in the framework of this work and his comments on the draft report of the thesis.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.12 Function conversions</td>
<td>28</td>
</tr>
<tr>
<td>3.2.12.1 Necessary Information</td>
<td>28</td>
</tr>
<tr>
<td>3.2.13 Occurrence Indicators</td>
<td>28</td>
</tr>
<tr>
<td>3.2.13.1 Necessary Information</td>
<td>28</td>
</tr>
<tr>
<td>3.2.14 Kind Tests</td>
<td>29</td>
</tr>
<tr>
<td>3.2.14.1 Necessary Information</td>
<td>30</td>
</tr>
<tr>
<td>3.2.15 Sequence Type Matching</td>
<td>30</td>
</tr>
<tr>
<td>3.2.15.1 Necessary Information</td>
<td>30</td>
</tr>
<tr>
<td>3.3 XDM construction, Typed and String Value determination</td>
<td>38</td>
</tr>
<tr>
<td>3.3.1 Requirements</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2 Issues</td>
<td>38</td>
</tr>
<tr>
<td>3.3.3 Suggested Solution</td>
<td>38</td>
</tr>
<tr>
<td>3.3.4 Implementation Details</td>
<td>38</td>
</tr>
<tr>
<td>3.3.4.1 Construction from PSVI</td>
<td>40</td>
</tr>
<tr>
<td>3.3.4.2 Typed Value Determination</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4.3 String Value Determination</td>
<td>41</td>
</tr>
<tr>
<td>3.4 Design Issues</td>
<td>32</td>
</tr>
<tr>
<td>3.4.1 Representation of the necessary information for UDTs</td>
<td>32</td>
</tr>
<tr>
<td>3.4.2 Limitations of the existing type system</td>
<td>32</td>
</tr>
<tr>
<td>3.4.3 Other requirements</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Xerces XML Schema API Implementation</td>
<td>32</td>
</tr>
<tr>
<td>3.6 Suggested Solution</td>
<td>33</td>
</tr>
<tr>
<td>3.6.1 Short Discussion on the Suggested Solution</td>
<td>33</td>
</tr>
<tr>
<td>3.7 Implementation Details</td>
<td>34</td>
</tr>
<tr>
<td>3.8 Other requirements and Issues</td>
<td>35</td>
</tr>
<tr>
<td>3.8.1 Review of the implementation of the affected XQuery features and operations</td>
<td>35</td>
</tr>
<tr>
<td>3.8.2 API extensions</td>
<td>35</td>
</tr>
<tr>
<td>3.9 Schema Import</td>
<td>35</td>
</tr>
<tr>
<td>3.10 UDT constructor functions</td>
<td>35</td>
</tr>
<tr>
<td>3.11 Validate Expression (ValidateExpr)</td>
<td>36</td>
</tr>
<tr>
<td>3.12 XDM construction, Typed and String Value determination</td>
<td>38</td>
</tr>
<tr>
<td>3.12.1 Requirements</td>
<td>38</td>
</tr>
<tr>
<td>3.12.2 Issues</td>
<td>38</td>
</tr>
<tr>
<td>3.12.3 Suggested Solution</td>
<td>38</td>
</tr>
<tr>
<td>3.12.4 Implementation Details</td>
<td>38</td>
</tr>
<tr>
<td>3.12.4.1 Construction from PSVI</td>
<td>40</td>
</tr>
<tr>
<td>3.12.4.2 Typed Value Determination</td>
<td>41</td>
</tr>
<tr>
<td>3.12.4.3 String Value Determination</td>
<td>41</td>
</tr>
<tr>
<td>3.13 ID, IDREF, IDREFs</td>
<td>42</td>
</tr>
<tr>
<td>3.14 Construction Modes (strip, preserve)</td>
<td>43</td>
</tr>
<tr>
<td>3.15 Node Kind Tests implementation</td>
<td>43</td>
</tr>
<tr>
<td>3.16 Functions and Operators</td>
<td>44</td>
</tr>
<tr>
<td>3.17 Testing Phase - Unit tests</td>
<td>44</td>
</tr>
<tr>
<td>4. User and developer guide</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Execution modes and command line interface</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1 Modes of execution</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1.1 Basic Mode</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1.2 Schema Aware Mode</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1.3 Validate Mode</td>
<td>46</td>
</tr>
<tr>
<td>4.1.1.4 Normal input Mode</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Examples</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Execution Modes - Java API</td>
<td>48</td>
</tr>
<tr>
<td>5. Stream Schema</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Foundations</td>
<td>49</td>
</tr>
<tr>
<td>5.1.1 Stream Definition</td>
<td>49</td>
</tr>
</tbody>
</table>
5.1.2 Known Issues in Data Stream Processing .................................................................49
5.2 The stream schema proposal ..........................................................................................50
5.2.1 Graphical Introduction to Stream Schema ...............................................................50
5.3 Changes in semantics and optimizations ......................................................................53
5.4 XML Schema extension ...............................................................................................54
6.Xerces for Streams ..........................................................................................................56
6.1 Introduction to Xerces .................................................................................................56
6.2 The Xerces extension ...................................................................................................58
6.2.1 The general approach ..............................................................................................58
6.2.2. Affected components ..............................................................................................59
6.2.2.1 Schema documents parser ..................................................................................59
6.2.2.2 XML scanner .........................................................................................................61
6.2.2.3 Validation ...............................................................................................................61
7.Results of the implementation of the XML Stream Schema proposal .................................64
7.1 Supported features overview ......................................................................................64
7.2 Limitations ..................................................................................................................64
8.Future Work ....................................................................................................................67
8.1 MXQuery .....................................................................................................................67
8.2 Stream Schema ...........................................................................................................67
Appendix .............................................................................................................................69
A.1 MXQuery Type System ...............................................................................................69
A.2 Sample Files ...............................................................................................................83
A.3 XML Schema for Streams - Formal Description ........................................................85
A3.1. XML Representation of Schemas ............................................................................85
A.3.2 Schema for schemas .............................................................................................88
A.4 Stream Xerces Sample Program .................................................................................93
Chapter 1

Introduction

1.1 Motivation

XQuery, the XML Query Language, W3C standard since 2007, is gaining momentum and becoming the established standard for querying XML data. XML Schema awareness is an optional feature of the XQuery language. Schema information can be exploited to facilitate and support predictability, early detection of errors, optimizations, special processing based on type and the validity of query input and results. The advantages of schema-awareness are described in detail in section 2.4. Currently, few XQuery implementations support the feature of schema-awareness and more importantly most if not all of them are commercial proprietary products.

At the same time, data stream processing has become a hot topic. One of the issues that the research community tries to deal with is the fact that streaming is dealing -by its very nature- with infinite data sources and in most cases a streamed element can be seen and accessed only once. [1] and [2] describe work on streaming extensions to XQuery(P) and XML Schema respectively. A window construct to execute queries on subsections of a stream and an extension to the XML Schema Data Model (XDM) to allow and mark possibly infinite elements has been described in [1], whereas [2] describes a proposal for XML Schema extensions to enable optimizations and early detection of problems arising with infinite data.

1.2 Objectives

This Master Thesis can be considered as consisting of two different parts.

The main goal of the first part is the integration of XML Schema-awareness in the MXQuery engine making it one of the first free and open-source implementations providing XML Schema support.

For the second part of the thesis, the modification of one of the publicly available tools for XML Schema validation in order to support the stream schema proposal described in [2] and the exploration of implications that the availability of such an extended tool has for query processing, streaming execution and optimizations can be considered as the main goal.

1.3 Outline

This report is divided in two main parts, in accordance with the two different parts of the thesis. Part One consisting of chapters 2-5 describes “XML Schema Support” in MXQuery. Chapter 2 is an introduction to the main technologies that constitute the foundations for this attempt of extending MXQuery with XML Support. Section 2.4 refers to the impact that the optional feature of
schema support has on XQuery queries. Section 2.5 is a brief introduction to the MXQuery engine. Chapter 3 describes the requirements for the integration of schema-related features in MXQuery and the implementation of each of the affected features. Section 3.1 is a short overview of the type system implementation in MXQuery, while 3.2 presents the requirements for the extension of MXQuery with schema support in terms of the implications that schema support has for different features of the language. The general structure of presenting the implementation details in Chapter 3, follows the steps: “Requirements and Issues” - “Suggested Solution” - “Implementation Details”.

Chapter 4 is actually a short user or developer guide for leveraging the functionality that the extended version of MXQuery offers. The way different modes of execution of the engine can be imposed either through the command line interface of the Java API of the engine are described in detail.

The second part consists of chapters 5 to 7. Chapter 5 sets the foundations for the second part. Chapter 6 is dedicated to the technical details of the second part. The beginning of the chapter is an overview of the architecture of the Xerces parser and validator that was extended to support streams. Chapter 7 summarizes the results of the second part with focus on the limitations and implementability issues that arose in the framework of this thesis.

Chapter 8 on “Future Work” for both parts completes the report.
Chapter 2

Foundations

2.1 XML Schema

XML Schema[3] (W3C Recommendation since May 2001), is a XML schema language; a language that can be used to express a schema: a set of rules to which an XML document must conform in order to be considered 'valid' according to that schema. XML Schema was also designed with the intent that determination of a document's validity would produce a collection of information adhering to specific data types. An XML Schema instance is an XML Schema Definition (XSD) and typically has the filename extension ".xsd".

The XML Schema specification defines 19 primitive data types:

- string
- boolean
- decimal
- float
- double
- duration
- dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth
- hexBinary
- base64Binary
- anyURI
- QName
- NOTATION

In XML Schema, there is a basic difference between simple types which cannot have element content and cannot carry attributes and complex types which allow elements in their content and may carry attributes. There is also a major distinction between definitions which create new types (both simple and complex), and declarations which enable elements and attributes with specific names and types (both simple and complex) to appear in document instances.

XML Schema also offers the possibility to define new types (simple and complex) based on other types (predefined, simple or complex).
Type definitions are either made globally or inside element declarations. They allow the user to define what an element of this type is composed of. Does it have attributes? Does it have child elements (complex types)? May it be empty? Or does it just contain character data (simple types)? If it is a complex type: Are there only elements as children or are they mixed with text? If it is a simple type: Is the character data constrained?

Declarations (element and attribute) declare where elements or attributes do appear. They are either used globally to be referenced inside or directly in type definitions. They mainly contain the following information: name, type, and namespace. For attributes, the usage is listed also to tell the user if this attribute can (optional), must (required) or must not (prohibited) be used in the element containing the declaration or reference. For elements, there may be additional information like its minimal (minOccurs) and maximal (maxOccurs) occurrence.

In addition, elements and types can have some restrictions on their extendibility. Types are all derived from an ur-type. Or more understandable, all types are a derivation of another. By following this derivation chain backwards, you will be able to reach the ur-type from any existing type. Derivations can be done in two ways:

- by restriction:
  - limiting some value to be one of a defined subset (enumeration)
  - by enforcing some pattern the value has to follow
  - removing some elements from the possible children
- by extension:
  - adding attributes or children
  - relaxing value restrictions

Attributes “block” and “final” used in a type's definition, specify whether a type can be used as base for restriction or extension.

Annotations can be placed in any schema construct to describe it more deeply in physical human language. An example of XML Schema document is given below:

```xml
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
  <annotation>
    <documentation xml:lang=en>
      Schema for publications
    </documentation>
  </annotation>
  <xs:element name=publications>
    <xs:complexType>
      <xs:element name=publication type=pubType maxOccurs=unbounded/>
      <xs:element name=authors>
        <xs:complexType>
          <xs:element name=author type=authorType maxOccurs=unbounded/>
        </xs:complexType>
      </xs:element>
    </xs:complexType>
  </xs:element>
  <xs:key name=AuthorAidKey>
    <xs:selector xpath=./publication/author/>
    <xs:field xpath=./@aid/>
  </xs:key>
</xs:schema>
```
The XML Schema given above makes use of all basic concepts:

• The root node is one of the globally defined elements, here <publications>.
• There are global type definitions (isbnType, pubType), as well as inner (anonymous) type declarations (for publications and publication type attribute).
• The isbnType and the publication type attribute type show derivations by restriction (the latter is a shorthand for restriction since neither <restriction> nor <extension> are given).
• Inside the author element a <publications/> element is used by reference.
• In the publications one can see the definition of a unique value: All <publication> inside <publications> must have a unique value for the ISBN attribute.
• There also exists an <xs:key> which can be referenced by <keyref refer=keyname>. Keys must also be unique. Keys are used in the example to reference the authors inside of a publication. Thus, each author has to be written only once in the authors element and can then be referenced by its aid attribute. This would also be the right approach to list the publications of an author (e.g. reference by ISBN).

An XML Schema Definition (XSD) is an instance of an XML schema written in XML Schema. An XSD defines a type of XML document in terms of constraints upon what elements and attributes may appear, their relationship to each other, what types of data may be in them, and other things. It can be used with validation software in order to ascertain whether a particular XML document is of that type. If an XML document and the documents describing its XML Schema are present, it is possible to validate the document. This means the parsed Infoset is checked if its items conform to the declaration rules in the given schema.

In the same step, the Infoset is enriched with additional information such as types and validation results.

The output of the validation is named a PSVI, a Post-schema-validation Infoset. The XML Schema documents from W3C [4] [5] define the validation rules for each schema component. The outcome of the validation is stored in the PSVI. All information added during the validation step of an item is declared in the PSVI Contributions section corresponding to the definition of the schema component in question.

XSDs were the first W3C-recommended XML schemas to provide a namespace and data type aware alternative to using XML's native Document Type Definitions (DTDs).

2.2 XQuery

XQuery[6] is a query language based on the structure of XML designed to query collections of XML data. It is a declarative, high level and side-effect free, functional programming language which is also Turing Complete.

XQuery is defined in terms of the XQuery 1.0 and XPath 2.0 Data Model (XDM), which represents the parsed structure of an XML document as an ordered, labeled tree in which nodes have identity and may be associated with simple or complex types.

In the XDM, a value is a sequence of items. The items in a sequence can either be nodes or atomic values. Atomic values may be integers, strings, booleans, and so on: the full list of types is based on the primitive types defined in XML Schema.

Variables, function parameters and return values can also be declared with an XML Schema type.

XQuery can be used to query (and apply transformations) on XML data that has no schema at all, or is validated against a XML Schema or Document Type Definition (DTD).
XQuery consists entirely of expressions. There are no statements, even though some of the keywords appear to suggest statement-like behaviors. XQuery expressions can be combined quite flexibly with other expressions to create new expressions.

XQuery uses XPath expression syntax to address specific parts of an XML document. It supplements this with a SQL-like "FLWOR expression" for performing joins. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN.

The language also provides syntax allowing new XML documents to be constructed.

The language is based on a tree-structured model of the information content of an XML document, containing seven kinds of node: document nodes, elements, attributes, text nodes, comments, processing instructions, and namespaces.

A XQuery query can be assembled from one or more fragments of XQuery code called modules. Each module is either a main module or a library module.

A main module consists of a Prolog (containing declarations e.g. variable and function declarations and imports) followed by a Query Body. A query has exactly one main module. In a main module, the Query Body can be evaluated, and its value is the result of the query. Library modules are modules that do not contain a Query Body. A library module cannot be evaluated directly; instead, it provides function and variable declarations that can be imported into other modules.

Most queries are written using a small number of constructs: FLWOR, if-then-else, XPath expressions and element constructors. They are presented briefly below by means of a number of small examples.

The simplest of the constructs, the element constructor:

```xml
<hello>Hello World</hello>
```

This is already an XQuery, it just returns an element with name “hello” and text content “Hello World”.

The next step is to use path expressions to use some values of another XML document:

```xml
<hello>{doc("hello.xml"):helloworld[1]}</hello>
```

This will still create an element with the name “hello”, but as content it uses the content of the first “helloworld” element of the document “hello.xml”. The reader can verify that this is just standard XPath usage. The curly braces denote, that the result of the contained sub-expression should be inserted instead of the expression itself.

Going a step further and using some conditionals:

```xml
if doc("test.xml"):isTrue eq "true"
  then <hello>{doc("hello.xml"):helloworld[1]}</hello>
else ()
```

As with all functional languages the last function defines the return value. Or in other words, the last returned value that is not consumed by some other function is the return value of the overall query. Therefore, the above query has the same output as the one before, only in the case where the content of the <isTrue> element of document “text.xml” is the string “true”. It is important, that the if-then-else in XQuery always needs an else-part. If there is nothing to do in the if- or else-expression, one must give the empty sequence.
Finally, the FLWOR expressions, the most powerful construct of XQuery:

```xml
let $persons := doc("persons.xml")/person
for $who in $persons
where $who/name eq "Peter"
order by $who/surname ascending
return $who
```

Every FLWOR expression must have at least one “for” or “let” and a “return” expression. “let” binds the returned sequence of its expression to the given variable. Note, that since XQuery is a declarative, functional language, variables can only be bound once. There is no possibility to update variable values, except for “for”.”for” iterates through the sequence returned by the expression after “in” and binds each element of the sequence iteratively to the given variable. This is done in the order in which the in-expression returns the elements unless an “order by” expression is given.

“order by” denotes at least one field which has to be used to re-order the element stream coming from the in-expression. “order by” has many options which will be skipped here for simplicity.

If needed, a “where” expression can filter out a subset of the sequence elements. If the “where” clause does not evaluate to true for an assignment, the “for” immediately restarts and takes the next element of the sequence. For each element passing the “where” clause, the “return” expression is evaluated. Therefore, the above query returns every <person> element of the “persons.xml” file whose <name> element has the content Peter sorted ascending by the content of its <surname> element.

XQuery (current version 1.0) was developed by the XML Query working group of the W3C. XQuery 1.0 became a W3C Recommendation on January 23, 2007.

XQuery 1.0 does not include features for updating XML documents or databases; it also lacks full text search capability. These features are both under active development for a subsequent version of the language.

### 2.3 XQuery Type System Quick Overview

The type system of XQuery is based on XML Schema. (For a more detailed presentation of XML Schema please refer to Section 2.1. Some of the XML Schema features are repeated in this section)

XML Schema facilities, including built-in types of XML Schema can be used to define types in XQuery (referred to as schema types). A schema type can be used as a type annotation on an element or attribute node.

Whenever it is necessary to refer to a type in an XQuery expression, sequence types are used (see SequenceType syntax). The term sequence type suggests that this syntax is used to describe the type of an XQuery value, which is always a sequence. (see [6] section 2.5, Types)

The following figure represents the hierarchy of Schema Types used in XQuery.
Schema types are divided in simple and complex types. In XML Schema, there is a basic difference between complex types which allow elements in their content and may carry attributes, and simple types which cannot have element content and cannot carry attributes. Simple types are further divided into atomic (have values that are regarded as being indivisible), union and list types in accordance with the corresponding XML Schema definitions.

Note: An atomic type is at the same time a sequence type and a schema type.

Another important distinction is that between primitive and derived types.

Primitive types are those that are not defined in terms of other data types, whereas derived types are those defined based on other types. Types (primitive or derived) defined in the XQuery specification are referred to as built-in data types, whereas derived types defined by schema designers individually are called user-derived types (please refer to Section 3.9 for more details on Schema Import Processing).

A list of the built-in data types is presented in Figure 2.3.

Derived simple types can be defined by derivation by restriction (constraining the base type's value space and/or its lexical space to a subset of those), by union or by list.

In the case of complex types, types can be defined either by extending or restricting other simple or complex base types (simple and complex content). Complex Type definitions in XML Schema also provide the option of specifying the content of elements as being mixed (character data can appear
between the child-elements), element only, and empty (no content, attributes only).

The following figure summarizes the main options for specifying User Defined Types (UDT).

Figure 2.2. UDTs
Figure 2.3: Built-in datatypes hierarchy
2.4 Impact of XML Schema on XQuery

The W3C Recommendation for XQuery describes the option of using and importing XML schemas in XQuery queries, allowing for the assignment of types to data and validation of both input XML and query results. This feature of schema-awareness provides the XQuery processor with information about definitions included in schemas, resulting in better optimized and tested queries. This information can be exploited to facilitate and support predictability, early detection of errors, optimizations, special processing based on type and the validity of query results. The advantages of using schema with XQuery queries are summarized below:

2.4.1 Predictability

The structure and data content of input documents that have been validated against a schema are predictable. Query writers can exploit this predictability and refer to specific elements of documents without having to check for their existence.

2.4.2 Type information for use in expressions

The schema provides type information to the query processor about the values and elements in the instance document. For example, in the case where a query returns sorted results (with “order by”) the query processor will provide different results based on the type information. (e.g. integer values are sorted differently than strings - 100 appears after 99 if sorted as integer values, whereas 100 would come before 99, if they were sorted as strings.

2.4.3 Identification of query errors

Schemas make it possible to the processor to determine the expected type of an expression during static analysis. Using schemas, errors in the query can be discovered, that were not otherwise apparent. Schemas can also make queries debugging faster and easier by providing more useful error messages. For example, without XML schemas nothing will be returned if an element name is misspelled or an invalid path is specified not providing enough information for debugging.

2.4.4 Query optimization

The more a processor knows about the structure of the input documents, the more it can optimize access to them. For example, if a schema is present, an expression such as catalog//number is a simple matter of looking at the grand-children of catalog and returning those named number. If no schema is present, every node of the document must be traversed to find number elements.

2.4.5 Special processing based on type

Type-related expressions, such as instance of and typeswitch, can be used on user-defined types in the schema.

2.4.6 Validity of query results

The results of a query can also be validated. This is especially useful if the XQuery delivers "intermediate" results to a "consumer" for further processing.
2.5 MXQuery

The MXQuery engine was developed as a lightweight XQuery engine which is able to run on mobile devices. In order to keep the footprint small, the MXQuery engine initially supported only a subset of XQuery. Most importantly, MXQuery did not implement the full typing system of XQuery. No user-defined types were supported (they were ignored) and only a subset of the atomic types (like xs:untypedAtomic, xs:string, etc.) were supported (internally). This reduces the memory and storage size of the engine dramatically. As part of this master thesis, the MXQuery engine will be extended with XML Schema support. Given that schema support and an extended type system is not considered to be so important for mobile devices and is also expected to introduce a significant performance overhead a lightweight version of the engine will still be maintained and offered together with the “new” schema-aware version.

The basic architecture of the engine is similar to the BEA XQuery engine and consists of a pull-based iterator model. This approach has big advantages: low main memory requirements, avoiding of materialization of intermediate results and modularity, and it supports lazy evaluation which is very important for infinite streams which already have been tested on the engine, see [1].

During parsing, the parsed query is translated into an iterator tree. Example:

```xml
for $x in doc("test.xml") / a / b
where $x = 25
return data($x/@c)
```

This simple query is translated into the tree in figure 2.4. On the top, there is the FLWOR iterator which contains operators for the FOR, WHERE and RETURN expression. The FOR iterator contains again an iterator (ChildIterator) which declares the binding items of the FOR. All operators of an iterator are again iterators. E.g., the IfThenElseIterator contains three sub-iterators as parameters. One for the condition, one for the THEN body and one for the ELSE body. The MXQuery engine contains for every major operation a certain iterator.

Iterators provide access to the sequence of XDM items corresponding to a specific query. The next() method of the Iterator returns the next item in the sequence. MXQuery uses Token objects (see ch.ethz.mxquery.xdm.Token and its subclasses) for the representation of items of the XQuery Data Model. There are different types of Tokens depending on the kind of item they represent or the type of value an item carries. Tokens are annotated with types -either XML Schema types or event types specific to the engine. e.g. START_TAG is an event type given to NamedTokens that represent the start tag of an element.

To get the result of the query, the top iterator (in this case the FLWOR iterator) must be iterated till it returns the END SEQUENCE Token. The result of the query is captured by serializing all produced Tokens of the FLWOR during the iteration. The complete iterator tree is executed lazily. When the next is invoked on the FLWOR iterator, the FLWOR iterator itself retrieves only as much information from its operators as it needs to compute the current Token.

A description of the MXQuery type system and the way the engine implements and represents the XQuery Data Model (XDM) can be found in sections 3.1 and 3.12 respectively.
Figure 2.4
Chapter 3

Requirements specification, Design and Implementation

This chapter provides information about the implementation of schema-related features in the MXQuery engine or what will be referred to as the “schema-aware version” of the engine. The chapter is structured as follows:

The first section is a short overview of the type system implementation in MXQuery. The second section is actually a requirements gathering/specification for the extension of MXQuery with schema support in terms of the implications that schema support has for different features of the language. In the remainder of the chapter, each section is dedicated to the implementation of each of the affected features. The general structure of presenting these implementation details is to first present the requirements followed by a short description of the suggested solution for meeting them and finally shortly describing the actual implementation. In cases where the requirements are only limited to what the XQuery specification describes, the description of the implementation of the specification is provided.

3.1 MXQuery Type System Revisited

This section briefly presents the type system that is used in the non-schema-aware version of MXQuery.

3.1.1 Encoding of types

MXQuery represents the XQuery type hierarchy by encoding types into integers (32 bits). Each type is represented by a combination of bits. Additional information necessary for representing sequence types, such as information related to node kind tests is encoded in a number of bits that are reserved for this purpose. Token-related event types are also encoded in the same way. (e.g. START_TAG, END_TAG etc) and are also part of the type system.

Bit and bit mask operations implement the basic operations on types in an efficient way (e.g. checking whether type A is derived from type B).

The MXQuery Type System API is presented later in this section.
For the implementation of type encoding the following conventions are used:

- All built-in types derived from a type have the same part as the parent type plus their own bits. In this way it’s easy to check if the type is subtype of another type.
- All built-in types from the same level and the same branch in the type hierarchy tree should have the same encoding length (e.g. all primitive 20 atomic types will have bit 1 at position 10).

**Reserved Bits**

All the information presented in this section corresponds to a description of the MXQuery Type System before the necessary extensions to deal with user-defined types.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDT</td>
<td>OI</td>
<td>OI</td>
<td>OI</td>
<td>E/S</td>
<td>NK</td>
<td>NK</td>
<td>NK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

31st UDT: User defined type bit (1 – user defined type, 0 – built-in type)
30-28 OI: Occurrence indicator bits (000 – undefined, 001 - 1 item, 010 – “?” , 100 – “*”,  011 –“+”,  111 – infinitive sequence )
27th E/S: bit for compatibility with existing event types (1 – START, 0 - END) only for element and document nodes
26 -24 NK: Node kind bits (000 – no node, 001 – element node, 100 – attribute node, 010 – text node)
These checks are done very often. To simplify process of node kind determination, each of the node kinds has a separate flag.

Note: The 32nd bit would be available if unsigned integers are used

For an complete documentation of the MXQuery Type System and Type Encoding please refer to the Appendix.

**Example: xs:integer**

Integer value: 12815
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Integer has 3 subtypes, so we reserve 2 bits for the encoding and 1 bit in front with value 1 for hierarchy separation

The parser uses these type constants while creating TypeInfo objects. TypeInfo object is used in node kind tests and sequence type verification.

The following section contains information about supported and not supported XQuery features related to the type system.
Not Supported Atomic Types:
  o xs:NOTATION, xs:NMTOKEN, xs:NMTOKENS, xs:ID, xs:IDREF, xs:ENTITY

Constructor functions for built-in atomic types have been implemented. These types are supported in
the appropriate iterators and functions.

Supported (implemented) Node Kinds:

<table>
<thead>
<tr>
<th>Name</th>
<th>Direct Constructor</th>
<th>Computed Constructor</th>
<th>Node Kind Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
<td>not supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Element</td>
<td>supported</td>
<td>supported</td>
<td>element name test supported; type name test not supported</td>
</tr>
<tr>
<td>Attribute</td>
<td>not supported</td>
<td>supported</td>
<td>attribute name test supported; type name test not supported</td>
</tr>
<tr>
<td>processing instruction</td>
<td>supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>comment</td>
<td>supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>not supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Node</td>
<td>not supported</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Item</td>
<td>not supported</td>
<td>supported</td>
<td></td>
</tr>
</tbody>
</table>

Note. Schema element and Schema attribute node kind tests are not supported.

Not Supported:
  o User defined (complex, union) types
  o Typed XML, schema imports

Type operators:

<table>
<thead>
<tr>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>castable as</td>
<td>implemented;</td>
</tr>
<tr>
<td>cast as</td>
<td>implemented;</td>
</tr>
<tr>
<td>typeswitch</td>
<td>implemented;</td>
</tr>
<tr>
<td>instance of</td>
<td>implemented;</td>
</tr>
<tr>
<td>Treat as</td>
<td>not supported; performs down-casting at compile time; it would make sense for static type checking;</td>
</tr>
</tbody>
</table>
Element and attribute content:

<table>
<thead>
<tr>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDATA section</td>
<td>implemented;</td>
</tr>
<tr>
<td>built-in character entities</td>
<td>implemented; (&lt; &gt; &amp; &quot; ' );</td>
</tr>
<tr>
<td>numerical character entities</td>
<td>not supported yet;</td>
</tr>
</tbody>
</table>

3.1.2 MXQuery Type System API

The MXQuery Type System, as described in section 3.1, is implemented by two classes: Type and TypeInfo, both contained in package ch.ethz.mxquery.model.types.

**Type**

The Type Class is the main class for the representation of types and their encoding as integer values. It contains the definition of the integer values for predefined types and the necessary masks. It uses a Hashtable data structure for the mapping between type names and integer representations. The necessary operations are implemented by means of the class methods which are briefly presented below:

**Method Summary**

- areComparable(int, int) : checks if two types are comparable
- createAttributeType(int) : sets the bit indicating attribute-test
- createTextNodeType(int) : sets the bit indicating text-test
- getAttributeValueType(int) : returns the type of an attribute value
- getEventTypeSubstituted(int): returns xs:integer for all types derived by xs:integer, xs:string for all types derived by xs:string and the same type for any other type
- getNumericalOpResultType(int, int): returns the result of a numerical operation (applying rules for type promotion and subtype substitution for numeric predefined types)
- getPrimitiveAtomicType(int) : returns the corresponding primitive atomic type
- getTextNodeValueType(int) : returns the type of an text node value
- getTypeFootprint(QName): return the integer value for a given QName representation of a type
- getTypeQName(int): return the corresponding QName for a given value
- highestOneBit(int): Find the highest set bit in value, and return a new value with only that bit set.
- initTypeQNameMappingStore(): initializes the Hashtable with the string, integer pairs (Qname,integer encoding)
- isAtomicType(int): checks if the actual type is an atomic type
- isAttribute(int): checks if the actual type is an attribute-test
- isEndElement(int): checks if the given type is an end type (END_TAG, END_SEQUENCE, END_DOCUMENT)
- isIdeedType(int): returns true if the given type has/need an id.
isNode(int): checks if the given type is a node
isNodeSubTypeOf(int, int): checks subtype relationship for nodes and items
isNumericPrimitiveType(int): checks if the given type is numeric
isStartType(int): checks if the given type is a start type (START_TAG, START_SEQUENCE, START_DOCUMENT)
isSubTypeOf(int, int): checks if type A is a subtype of type B
isTextNode(int): checks if given type is a text-test
isTypeOrSubTypeOf(int, int): checks if type A is the same type or a subtype of type B
isUserDefinedType(int): checks if a type is a UDT
setTypeBit(int, int, char): sets a specific type bit
typePromoteableTo(int, int): checks if type A can be promoted to type B (type promotion and subtype substitution rules for predefined atomic types)

TypeInfo

Several node kind tests contain a name part (e.g. element and attribute tests). This additional information cannot be encoded by the integer representation used in the Type class. The TypeInfo class is used in the case of node kind tests and sequence type matching, where information about names also needs to be stored.

3.2 Implications of UDT support for other XQuery Features

This section presents the results of an attempt to enumerate all the XQuery features that are or might be affected by the integration of UDT and schema-awareness in the MXQuery engine. For each feature a reference is made, to type specific information that should be available (e.g. the information about whether a type is an atomic type)

3.2.1 Type Annotations

Element and attribute nodes in an XDM instance have a type annotation (see type-name property of XDM and dm:type-name Accessor) If the XDM instance was derived from a validated XML document the type annotations of the element and attribute nodes are derived from schema validation.

3.2.1.1 Necessary Information

The name/encoding of the schema type of a node should be available.

3.2.2 Typed Value

The typed value of a node (XDM Accessor dm:typed:value) is a sequence of atomic values and can be extracted by applying the fn:data function to the node. The typed value can be specified by applying specific rules for each node type. (see [6] section 2.5.2, Typed Value and String Value and [9] )
3.2.2.1 Necessary Information
(see rules [6][9])

The name/encoding of the schema type of a node should be available.
All the information presented in Figure 2.2 should be available for the typed-value to be specified.
The “nilled” property of a node should be also be known.
For more information about “typed-value” please refer to section 3.12, on XDM construction.

3.2.3 Automatic Type Conversions Subtype substitution

Functions and operators that expect a value of a particular type also accept a value of one of its derived types (derived by restriction for simple types, and either by restriction or by extension for complex types).

3.2.3.1 Necessary Information
Derivation relationships between types (base type and type hierarchies).
Type of derivation (restriction/extension vs union and list)

3.2.4 Type promotion

During the evaluation of function calls, order by clauses and operators accepting numeric and string operands, an atomic value might be promoted to another type.

3.2.4.1 Necessary Information
Indication about whether a type is atomic.
The name/encoding of the atomic schema type of a node should be available.

3.2.5 Atomization

Atomization is a process that occurs when a function or operator expects an atomic value and receives a node instead.

Atomization is used in processing the following types of expressions:

- Arithmetic expressions
- Comparison expressions
- Function calls and returns
- Cast expressions
- Constructor expressions for various kinds of nodes
- order by clauses in FLWOR expressions
Atomization involves extracting the typed value of elements or attributes to return atomic values.

### 3.2.5.1 Necessary Information
(see typed values):

All the information presented in Figure 2.3 should be available for the typed-value to be specified.

### 3.2.6 Cast

XQuery provides a *cast* expression that creates a new value of a specific type based on an existing value. Casting involves atomization and is supported only for atomic types (either primitive or non-primitive derived by restriction).

### 3.2.6.1 Necessary Information

Indication about whether a type is atomic (simple type derived by restriction from an atomic type).

### 3.2.7 Constructor functions

A constructor function is a function used to construct atomic values for given types. A construction function is implicitly defined for every atomic type; both predefined and UDT (simple, atomic Types) in the in-scope schema definitions. Calling a constructor function includes arguments atomization. The semantics of the constructor function call `$T(arg)` are defined to be equivalent to the expression `(($arg) cast as T?)`.

#### 3.2.7.1 Necessary Information

Indication about whether a type is atomic.

### 3.2.8 Castable operator

The castable expression is used to determine whether a value can be cast to another specified atomic type (predefined or UDT).

#### 3.2.8.1 Necessary Information

Indication about whether a type is atomic
Name/encoding of the atomic type
3.2.9 Treat as

“Treat as” is an expression that can be used to modify the static type of its operand. The dynamic type is not affected (unlike cast). During the expression evaluation, rules for Sequence Type Matching (see below) are applied.

3.2.9.1 Necessary Information

(see Sequence Type Matching)

3.2.10 Typeswitch

The typeswitch expression chooses one of several expressions to evaluate based on the dynamic type of an input value. Rules for Sequence Type Matching are applied.

3.2.10.1 Necessary Information

(see Sequence Type Matching)

3.2.11 Instance of

The boolean operator instance of returns true if the value of its first operand matches the SequenceType in its second operand, according to the rules for SequenceType matching; otherwise it returns false.

3.2.11.1 Necessary Information

(see Sequence Type Matching)

3.2.12 Function conversions

The function conversion rules are used to convert an argument value or a return value to its expected type (declared type of the function parameter or return, see [6], Section 3.1.5 Function Calls) Function conversion rules also involve atomization.

3.2.12.1 Necessary Information

(see rules [6])

3.2.13 Occurrence Indicators

3.2.13.1 Necessary Information

(see Necessary Information for Atomization)
3.2.14 Kind Tests

Kind Tests is a way of matching sequence types with sequences. (see also Sequence Type Matching below). When schemas have been imported and UDTs are also supported the following tests are affected:

− Element and Attribute tests
  An ElementTest is used to match an element node by its name and/or type annotation
  An AttributeTest is used to match an attribute node by its name and/or type annotation
− Schema – Element and Schema -Attribute Tests

A SchemaElementTest matches an element node against a corresponding element declaration found in the in-scope element declarations.
A SchemaAttributeTest matches an attribute node against a corresponding attribute declaration found in the in-scope attribute declarations.

The rules corresponding to these tests are presented in detail in [6], Section 2.5.4 Sequence Type Matching.
The following table presents a number of examples for the tests described above, summarizing the main different cases and rules:

<table>
<thead>
<tr>
<th>Example</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema-element (myName)+</td>
<td>On one or more elements whose name is myName, or whose name is the same as an element in the schema-defined substitution group of myName. myName must be the name of a global element definition in an imported schema, and the element being tested must have been validated against the type defined for myName in the schema.</td>
</tr>
<tr>
<td>Schema-element (my:myName)</td>
<td>One element whose qualified name is equal to my:myName, that has been validated using a global element declaration in the in-scope schema definitions -ISSD (as above)</td>
</tr>
<tr>
<td>element(*, my:myType)</td>
<td>One element whose type annotation is my:myType or a type derived from my:myType</td>
</tr>
<tr>
<td>element (my:myName,my:myType)</td>
<td>One element whose qualified name is equal to my:myName whose type annotation is my:myType or a type derived from my:myType, that is in the ISSD</td>
</tr>
<tr>
<td>attribute(abc) schema-attribute(abc) attribute(*, my:myType) attribute(abc, my:myType)</td>
<td>These sequence types match attributes in much the same way as the corresponding element() variants. It's unusual for schemas to define many top-level attribute definitions so some of these forms are rarely used. Moreover, there's no equivalent of substitution groups in the case of attributes.</td>
</tr>
</tbody>
</table>
3.2.14.1 Necessary Information

Element/Attribute name
Name/encoding of the schema types involved
Derivation relationships between types (base type and type hierarchies).
Occurrence indicators (e.g. when matching a SequenceType and a Value)
Substitution groups information
(Global) element/attribute definitions in imported schemas

3.2.15 Sequence Type Matching

Sequence type matching is the process of determining whether a sequence of items matches a specified sequence type, according to the rules specified in [6], Section 2.5.4 Sequence Type Matching. Several kinds of expressions perform sequence type matching, such as the instance of and typeswitch expressions. Sequence Type matching relies on the derivation relationship and Node KindTests.

3.2.15.1 Necessary Information

Name/encoding of the schema types involved
Derivation relationships between types (base type and type hierarchies).
Type of derivation (restriction/extension vs union and list)
Occurrence indicators (e.g. when matching a SequenceType and a Value)

3.3 Type System extension for UDT support

This section provides a summary of the requirements for integrating UDT support – and eventually schema-awareness in the MXQuery processor. The requirements are analyzed along three “dimensions”; information that should be available in a “schema-aware” Type System, Design Issues for the extension/enhancement of the Type System with this additional information, necessary API and operations extensions.
3.3.1 Necessary Information

According to the results of the analysis in the previous section, the necessary information that should be stored in the type system for UDT support is:

- Name/encoding of the imported schema types
- Element/Attribute names involved in Kind Tests
- Derivation relationships between types (base type and type hierarchies).
- Simple vs Complex types
- Atomic vs Union vs List Types
- Type of derivation (restriction/extension vs union and list)
- Occurrence indicators
- Content type of complex types

**Note:** Part of the information necessary for Kind Tests (e.g. substitution groups) can be provided by accessing the PSVI and does not need to be stored and supported by the Type System.
3.4 Design Issues

3.4.1 Representation of the necessary information for UDTs
The MXQuery Type System needs to be extended in order to store the additional information about UDTs and also provide a way to access this information efficiently (e.g. check about subtype relationships). The following factors should be taken into account in designing a solution for UDT support in the MXQuery Type System:

3.4.2 Limitations of the existing type system
The extensibility of the integer representation of predefined types with respect to encoding UDTs is rather limited. There are only 2 unused bits (3 if unsigned integers are used). A decision of moving the occurrence indicator bits to the TypeInfo class, would set 3 more bits free, leading to a total of 6 available bits and a value space of $2^6$ values for representing complex types and all the information related to them. Moreover, the hierarchy of types is not known in advance – as in the case of predefined types- making it difficult to present type hierarchies using the same schema used so far.

3.4.3 Other requirements
- Extensibility
- Performance Issues
  - Schema parsing is an expensive operation. Optimizations and design decisions should be made to avoid unnecessary repetitive parsing of schemas that have already been processed.
- Integration of Xerces – Mapping of Xerces Type Representation to MXQuery Type System
- Correct handling and association of Schema Import with query modules

3.5 Xerces XML Schema API Implementation
Xerces implements the XML Schema API specification that defines an API for accessing and querying the Post Schema Validation Infoset (PSVI) and interfaces for loading XML schema documents and querying against them. All schema components and their properties (e.g. facets) are represented and methods are provided for accessing the components information and performing useful operations (e.g. checking subtype relationships, derivation hierarchies etc.). All the necessary information for UDT support in the type system, is a small subset of the information that the XML Schema API provides.

The XML Schema API defines the xSObject interface which is implemented by all the XML Schema component model classes. Types are represented by instances of classes (see TypeDefinition) that also implement the xSObject interface. For the representation of type hierarchies each TypeDefinition instance contains a reference to an instance corresponding to its base class. Hence, for checking subtype relationships and derivations navigating through this type hierarchy -actually a linked list of types- is necessary.
3.6 Suggested Solution

A solution that meets the requirements that have been analyzed in the previous sections is described in this section.

The MXQuery type system will be enhanced with the addition of a dictionary of types, used to store information about UDTs. The entries of the dictionary of types are XObject instances, corresponding to XML Schema components as specified by the XML Schema API and implemented by Xerces. The dictionary should be filled with entries each time a schema document is imported and parsed, and associated with each separate module that imports the corresponding target namespace (Any module that needs to refer directly to type definitions contained in a schema, must import the schema explicitly in its prolog - even if this schema is imported by another imported module). The following section on “Implementation Details” contains more information about the type dictionary.

The connection with the rest of the type system will be accomplished as follows:

The $31^{\text{st}}$ bit of the type integer representation serves as a flag for identifying UDTs. When this bit is set, the reserved bits maintain their semantics, whereas the rest of the bits (1-23) should represent the indexed position of the UDT in the dictionary. Hence, the extended type system should provide a mapping of integer representation of types to actual dictionary entries (UDT), a mapping of UDT names to integer representations (the index part of the representation) and finally a mapping of UDT names to actual UDT dictionary entries.

The dictionary should thus provide the following “lookup” facilities:

- Find UDT definition based on name
- Find UDT definition based on position (index) in the dictionary
- Find entry position based on UDT name (to be used for creating a unique integer representation)

The methods provided by the XML Schema API should be sufficient for accessing all the necessary information stored in the dictionary.

Note: The MXQuery Type System will have to be revisited again for the needs of the construction of the XQuery Data Model. Please refer to section 3.7 and the Appendix for a complete documentation of integer encoding of types in MXQuery.

3.6.1 Short Discussion on the Suggested Solution

The decision on the suggested solution was made after considering the advantages and disadvantages of different possible solutions and the trade-offs involved.

From a performance perspective, an ideal solution would be based on the idea of extending the existing MXQuery Type System in order to represent user defined types and all the necessary relationships between types. This would have the significant advantage that all the operations needed (especially subtype relationship check) could be performed by means of binary operations. At the same time however, the mapping between the MXQuery Type System representation and the Xerces Type System classes would be more complex to implement. The main problem of such a solution has to do with its feasibility. Given the lack of free space in the 32-bit integer representation that has been described in previous sections, and the fact that type hierarchies consisting of user defined types can be of arbitrary length and complexity and not known in advance, a solution based on the 32-bit integer representation used by MXQuery seems to be infeasible or difficult to implement in a “clean cut” and elegant way.
The suggested solution, on the other hand, does not seem to be so attractive with respect to performance. It is clear that the way Xerces deals with the representation of types and especially relationships between types introduces a significant performance overhead compared to the really efficient solution supported for built-in types by MXQuery. As it has been already described in the previous section, each type a subtype relationship check is to be performed, visiting an arbitrary number of nodes of a linked list is necessary. The main “virtue” of the suggested solution lies in the fact that it provides a very clear separation of handling built-in and user defined types as it has been described above. Every time a user defined type is encountered all the operations and checks are “delegated” to Xerces. Thus, no major changes have to be done in the MXQuery type system and using Xerces really offers a lot of features “for free”. The results of performance experiments about the actual performance overhead that the suggested solution for the Type System introduces, would be very interesting to analyze.

3.7 Implementation Details

Class TypeDictionary (ch.ethz.mxquery.model.typesTypeDictionary) implements the Type Dictionary that MXQuery uses for UDT support.

An issue that was taken into account when designing the dictionary was the question about whether a schema-aware version of the MXQuery engine could run on CLDC. The fact that there is no schema-validating parser for CLDC available is rather limiting for having a CLDC schema-aware version in the near future. However, the implementation of new engine components like the Type Dictionary should also not be very limiting with respect to CLDC.

The TypeDictionary class consists of 3 Hashtables facilitating the implementation of the dictionary itself along with the necessary lookup operations that have been identified by the requirements. One of the hashtables contains the global XML Schema type, element and attribute definitions as values (as provided by the XML Schema API when a schema document is parsed) and its keys are the names of UDTs in the form {namespace uri} typeName, [namespace uri] elementName or (namespace uri) attributeName, for types, elements and attributes respectively. The other 2 hashtables are used to allow performing lookups based on the type name and integer representation of a user defined type, mapping integer representation (actually dictionary index position) to type name and vice versa respectively. (A fourth hash table is used to store the names of atomic UDTs and the generation of constructor functions for them, in the special case of schemas being pre-imported. (Please refer to section 3.5 on UDT constructor functions for more details).

The decision was made to “store” the dictionary at the “Global Context” level of the whole engine.(see Context.globalContext). Thus, schema documents associated with a specific target namespace are being parsed only once. Once, the schema components of a target namespace have been imported any further imports of schemas for this namespace are ignored.

In order to make sure that each XQuery module can only refer to types and schema components associated with the namespaces it has explicitly imported, each model holds a list of target namespaces it is aware of, at the module level.

The design decision of storing the type dictionary at the global level, introduces a number of thread safety and synchronization issues which can appear during parallel query processing. These issues are a resolved by explicit locking resources if necessary. For example, java.util.Hashtable is a thread safe structure, however there as still cases were race conditions can appear if methods where calls are made
to both the put() and get() methods of Hashtable. Such methods are declared as “synchronized”.

The signatures of the methods of class Type had to be modified, in cases where the dictionary also had to be available.

The code for the whole system had to be refactored accordingly.

### 3.8 Other requirements and Issues

#### 3.8.1 Review of the implementation of the affected XQuery features and operations

The implementation of the affected XQuery features and operations – as presented in Section 3.2 - should be reviewed. These features should be updated and enhanced, implementing the full set of rules related to these features as defined by the XQuery recommendation and also integrating necessary API extensions.

#### 3.8.2 API extensions

A list of the methods that were added to the Type.java is given below:

- `getUDTName(QName, TypeDictionary)`: Support method used by method `getTypeFootprint`, to transform the type name provided as QName in the appropriate form for dictionary lookups - `{namespace uri} typeName

- `isUDTAtomic(int, TypeDictionary)`: checks whether the given UDT is atomic. Used by method `isAtomicType`.

- `isUDTsubtypeof(int, int, TypeDictionary)`: checks if type A is a subtype of type B, where at least A is a UDT. Used by method `isSubTypeOf`.

### 3.9 Schema Import

The MXQuery parser (ch.ethz.mxquery.parser.Parser) has been updated to implement schema import processing. The implementation of this feature is fully compliant with the XQuery specification.

The In Scope Schema Definitions (ISSDs) are updated every time a valid schema is imported (the Type Dictionary is filled with entries), prefixes are bound to targetNamespaces and errors are raised if necessary in accordance with the specification.

### 3.10 UDT constructor functions

Constructor functions for atomic user-defined types are supported in the schema-aware version of MXQuery.

Upon parsing a schema document – either because a SchemaImport statement has been encountered or because schemas have been pre-imported - UDT constructor function signatures for all user-defined atomic types are added to the function gallery at the root context level (see Context.rootContext).
The main concept behind the implementation of constructor functions for UDT is to reuse the already existing constructor function implementations that are available for the built-in atomic types and modify them in order to make use of the information about restricting facets that have been defined in schema. More specifically all built-in atomic type constructor functions (see package ch.ethz.mxquery.functions.xs) have been slightly modified. They extend XSConstructorIterator which contains fields for specifying the facets and provides methods for accessing them. Thus, every time a atomic UDT is identified when parsing a schema document, a new function signature is simply created for this type and added to the function gallery. The function signature is associated with the constructor function implementation corresponding to the base type of the atomic UDT. When the function is called and the corresponding function implementation (Iterator) is used, a check is performed at the Iterator, to identify which facets need to be checked and applied to the argument of the constructor. For example, all constructor functions for atomic types derived by xs:string make use of the xs:string constructor (ch.ethz.mxquery.function.xs.XSString) and use the setFacetsList() method to pass the list of facets to the Iterator.

A final issue worth mentioning has to do with the point in query processing where the new function signatures are added to the function gallery. Since schema processing can be triggered at two different points in time; schema preprocessing when the engine is initialized and instantiated (see command line arguments and modes of execution – Section 4.1) or during the parse phase when a schema import statement is encountered, updating the function gallery of a module with UDT constructor functions is treated differently in each case.

The former case is slightly more complicated since the context containing the function gallery to which the signatures will be added has not been created when schema documents are parsed. Hence, the process of adding the function consists of two main stages. Initially, when the schema is parsed and an atomic type is encountered, the description of this type (as it is provided by Xerces – see XSTypeDefinition) is appended to a list that is created for the target namespace of the schema that is being parsed. When schema parsing is complete, this list is added to a hash table held at the global type dictionary using the target namespace as a key. Hence, at a later stage, during the parse phase and more specifically at a SchemaImport statement, a lookup is performed in the hash table for the namespace that is being imported, the list, if any, is retrieved and constructor functions for all the items (types) in the list are added to the function gallery.

Finally, in the latter case, both parsing the schema documents and adding the constructor function signatures to the function gallery are triggered when a SchemaImport statement is encountered in a query module.

3.11 Validate Expression (ValidateExpr)

The validate expression is supported in the schema-aware version of MXQuery. ValidateExpr has been implemented in compliance with the XQuery Recommendation.

Validation is the cornerstone of the validate expression and schema-aware XQuery in general. One of the major issues that had to be resolved, was the solution that should be given for validation. Given the limited time that was available in the framework of a master thesis, the decision was made to use one of the available validator tools that would also provide support for PSVI access. Xerces and XMLBeans, both open source Apache projects, were considered and evaluated.
Initially, XMLBeans, offering a validator and PSVI provider accessible through a Stax interface, seemed to be the strongest candidate, since the “streaming” nature of Stax, would fit and integrate more naturally in the Iterator-model-based MXQuery engine. However, an evaluation of the current version of XMLBeans revealed a number of bugs in validation and PSVI construction. As far as performance is concerned, the lack of performance experiments comparing validator tools, did not allow for providing strong arguments for the performance advantages of XMLBeans. One last point that could be considered as a disadvantage of the XMLBeans project, is the fact that validation is mainly a “byproduct” of the project. As a result documentation is rather poor and using the XMLBeans distribution as part of a future MXQuery distribution would make it necessary to carry along a substantial amount of code for features that were actually unnecessary.

Xerces, on the other hand, has the main disadvantage that the input has to be serialized before being given to the validator. Still suffering from some bugs at less critical points, the strong point of Xerces is also the fact that it implements the very well designed and defined XML Schema API which provides access to all the necessary information for the needs of schema-awareness in XQuery. Table 3.1 summarizes the results of the validator tools evaluation.

<table>
<thead>
<tr>
<th>Xerces vs XMLBeans</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+) Implements XML Schema API</td>
</tr>
<tr>
<td>(-) To-be-validated input has to be serialized</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Claim that XMLBeans performs better mostly an intuition</td>
</tr>
<tr>
<td>No performance experiment results available</td>
</tr>
</tbody>
</table>

Table 3.1 – Summary of validator tools evaluation results

Xerces provides access to the PSVI through its own SAX Parser implementation which also implements the so called PSVIPProvider interface. (see [10] ). The interfaces of a number of handlers for SAX events have been implemented. The SAX events are handled and transformed to the creation of the necessary tokens for the correct XDM generation in MXQuery. Within the scope of a method corresponding to an event, the PSVIPProvider offers PSVI information which is used for the annotation of tokens accordingly. Unfortunately materialization cannot be avoided, since in many cases PSVI information is available only when the end of an element is encountered by SAX. Thus, the startElement method creates a new token, which can only be annotated within the respective endElement method and after all the events for children elements have been handled. For more information of XDM construction from the PSVI please refer to the following section.
3.12 XDM construction, Typed and String Value determination

MXQuery uses Token objects (see ch.ethz.mxquery.xdm.Token and its subclasses) for the representation of items of the XQuery Data Model. There are different types of Tokens depending on the kind of item they represent or the type of value an item carries. For example items that have a name such as element nodes (start or end tags) are represented as NamedToken (or NamedToken subclass) instances and an item with a double value is represented with an instance of DoubleToken.

Tokens are annotated with types -either XML Schema types or event types specific to the engine e.g. START_TAG is an event type given to NamedTokens that represent the start tag of an element.

Schema-awareness and the support for user-defined types in MXQuery introduces a number of issues mainly with respect to type annotation. A really critical point in the XDM representation is the construction of XDM from PSVI information and the correct implementation of the typed-value XDM accessor.

3.12.1 Requirements

The rules for construction from PSVI and typed-valued determination were followed as described in [6], Section 2.5.2 and [9], Section 3.3.1. The kind of type (simple or complex), type of content for complex types (mixed, element only, simple), kind of simple type (atomic, list, union), the “nilled” property of a node and in some cases the schema-normalized-value of an item constitute the necessary information for the implementation of construction from PSVI and typed-value determination.

3.12.2 Issues

The integer encoding of a user-defined type also serves as an index to an entry in the type dictionary which stores all the necessary information about a type definition. Thus, information about the kind of type and content can be provided by accessing the dictionary. On the other hand, the “nilled” property or the schema-normalized value can only be provided by the PSVI. Hence, simply annotating tokens with the integer representation of a type is not sufficient. Moreover, the support for typed XML and the need for typed-value determination in this case, implies that a NamedToken should be annotated not only with the event type (e.g. START_TAG) but also with a XML Schema type that corresponds to the element in question.

3.12.3 Suggested Solution

When XDM is constructed from PSVI information, tokens representing element nodes are annotated with the event type (e.g. START_TAG, END_TAG) and the XML Schema type available for the PSVI in the case of START_TAG. For performance reasons both even type and schema type information is carried at the same integer field (see Token.eventType) and the necessary information can be extracted with the application of binary masks. The code of the whole engine has to be refactored accordingly.

The integer encoding of some events needs to be slightly modified in order to avoid overlaps between event types and schema types representation. For example the representation of START_TAG needs to change so as to be out of the schema types representation value range and the element_node token type has to be eliminated and conceptually “merged” with START_TAG. (The name START_TAG should also be reviewed and altered to be consistent with its new meaning). Similarly the attribute_node token...
type has been eliminated and conceptually merged with the AttributeTest node kind test (see TYPE_NK_ATTR_TEST integer encoding). For the representation of the “nilled” value the 23rd bit of the integer representation of the type (eventType) of the Token is used. When the bit is set, the “nilled” value is set to true. A field for carrying the schema-normalized-value has been added to token. The field is assigned a value only when this is necessary for typed valued determination (simple types, complex types with simple content, anySimpleType and anyAtomicType).

Figure 3.1 presents the updated positions of the reserved bits in the integer representation of types.

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>UDT</td>
<td>NK</td>
<td>E/S</td>
<td>NK</td>
<td>NK</td>
<td>NK</td>
<td>NIL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.1 – Reserved Bits (updated)**

31st UDT: User defined type bit (1 – user defined type, 0 – built-in type)
27th E/S: bit for compatibility with existing event types (1 – START, 0 - END) only for element and document nodes
28-24 NK: Node kind bits (00000– no node, 11000 element node/START_TAG, 00100 – attribute node/AttributeTest, 00010 – text node)
23: NILLED property

Figure 3.2 shows the integer representation of a the type annotation for an NamedToken which corresponds to a START_TAG and is of type xs:NCName along with the binary mask that needs to be applied in order to extract the actual type annotation.

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**START_TAG (red) + xs:NCName (blue)**

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Mask**

**Figure 3.2 – START_TAG + xs:NCName**

For a complete description of the updated MXQuery Type System please refer to the Appendix.
3.12.4 Implementation Details

3.12.4.1 Construction from PSVI

Class ValidatedSaxImportAdapter is used to validate input and transform SAX events to MXQuery XDM representation, namely tokens, annotated with types. ValidatedSaxImportAdapter uses the SAX Parser implementation that is “shipped” with the Xerces distribution. The Xerces SAX Parser also implements the PSVIProvider interface whose methods provide access to all the PSVI information items and properties needed for token type annotation and typed value determination.

The process of transforming SAX events to Tokens is described below:

The bodies of the methods-handlers of SAX events create the necessary tokens and append them in a list (Vector) of tokens. At the end of this process, an iterator over this list is retrieved and provides access to the validated input based on an Iterator model. The logic implemented in most of the methods is really straightforward. For example the startDocument() handler leads to the creation of a Token annotated with a START_DOCUMENT event type, processingInstruction() creates a ProcessingInstrToken (the MXQuery Token for processing instruction representation) etc. The most interesting part lies in the startElement() and endElement() methods of the SAX handler. Within the body of the startElement() method a new NamedToken is created, given the START_TAG event type and appended to the list of tokens. The list of attributes given by SAX is then visited and an attribute token of the type suggested by the PSVI is created and appended to the list. MXQuery provides token types for attributes for the built-in types. Hence, in the case when an attribute of a user-defined type is encountered, its base built-in type is identified and a token corresponding to that type is created. As far as the type annotation of NamedTokens is concerned, only tokens with event type “START_TAG” are annotated. This can only be done when the end of an element is found (endElement() method). At that point, the PSVI for the element in question is accessed, the type of the element is provided, its integer representation for the MXQuery type system is retrieved and the token – which already resides in the list – is annotated with the type by means of a binary OR operation adding the schema type information to the already existing event type information. A special case for type annotation, is that of list types consisting of items of type union. The problem with this case, is that each item of the list has a different type, which can only be given by the PSVI and at the same time the integer representation of types and event types is rather limited in order to carry the information about all the items present in a list. The solution for this special case, is to introduce a field in NamedToken to carry the types of list items when necessary. This field is actually a list of short numbers –Xerces internal representation of types uses short integers. Thus, a mapping between Xerces types to MXQuery types is also necessary. This whole procedure of additionally storing a list of numbers and then transforming them to MXQuery representation, introduces an overhead, which however should be relatively rare, being necessary only at the exceptional case of a list type with items of type union. The validity property of the PSVI is always taken into account before annotating tokens and the rules for XDM construction are strictly followed.

Notes: Since input that needs to be validated has to be serialized before being passed over to Xerces, new node ids are created and assigned to all the tokens.

Another point worth mentioning has to do with anonymous types. Anonymous types appear for the first time during type annotation and PSVI access, since when a schema is parsed only global element, attribute and type declarations are accessible. However, some tokens need to be annotated with anonymous types, which also need to be declared at the dictionary of types. The decision was made to
insert anonymous types in the dictionary only when necessary, meaning only when a token needs to be annotated with an anonymous type. The process is exactly the same as with global type declarations. Xerces automatically creates names for anonymous types. These names are only used when a type is inserted at the dictionary and in cases where a lookup based on name is executed. However, since most of the type related operations are based on the integer representation of a type, this information is sufficient for retrieving all the necessary properties of a type definition, without caring about the proprietary way that Xerces uses to name anonymous types.

3.12.4.2 Typed Value Determination

The determination of the typed-value of nodes/tokens lies at the heart of the XQuery language and the MXQuery engine. The DataValuesIterator (ch.ethz.mxquery.function.fn.DataValuesIterator) is responsible for delivering the typed value of a node. The DataValuesIterator had to be modified in order to operate correctly for typed XML. The rules for typed value determination were applied and integrated in the previous version of the iterator. In summary, the getValue() method which is used for performing atomization, has been extended. The type and content type of items is examined and the rules for typed value determination are applied accordingly. The most interesting case has to do with typed values of tokens with simple type or complex type with simple content where the constructor function of the corresponding atomic type actually needs to be applied to the value of a token. The getValue() method has been modified to return the appropriate iterator implementing the constructor function. Thus, all the necessary facets of a type are applied and checked and the result token(s) carry the correct type information. Again, typed value determination for list types with items of type union, request special treatment. In this case, the value of the text node that the element or attribute contains is split into string tokens, the constructor corresponding to the actual item type is applied and a SequenceIterator “holding” the tokens is returned. If untyped XML is given as input to the DataValuesIterator the behavior is the same with the previous non-schema aware versions of the engine and has remained unaffected by the modifications and extensions for typed XML.

3.12.4.3 String Value Determination

The string-value accessor is implemented in MXQuery by the FNString iterator. The introduction of typed XML does not affect the computation of the string value, with the exception of nodes with simple type or complex type with simple content where the schema normalized value should be returned. The main issue for the implementation of the string value accessor in the schema-aware version of the engine, was the fact that the FNString iterator was based on the DataValuesIterator. In the schema aware version, FNString still uses the DataValuesIterator but two different modes have been introduced to the latter. When the flag “fnData” is set to true – which is the default behavior – the DataValuesIterator returns typed values. If the flag is set to false, as in the exceptional case of FNString, the DataValuesIterator returns string values. A common example of difference between the actual value that should be returned as string-value and typed-value refers to the application of the two accessors on an element of complex type with element-only content. The string-value of this element is the concatenation of its text nodes in document order, whereas the typed value is undefined, and an attempt to request the typed-value of such an element should lead to raising an error. These two different modes of operation of the DataValuesIterator make sure that the two accessors are handled correctly.
3.13 ID, IDREF, IDREFs

IDs and IDREFs identify elements in a unique way within an XML document and create references to those elements respectively. Typically, an attribute is used as an ID to uniquely represent the element that carries it. The value of that ID attribute must be a valid NCName (an XML name with no colon). Attributes named xml:id (in the http://www.w3.org/XML/1998/namespace namespace) are always considered to be IDs. Attributes with other names can also be considered IDs if they are declared to have the built-in type xs:ID in a schema or DTD.

The type xs:IDREF is used for an attribute that references an xs:ID. All attributes of type xs:IDREF must reference an ID in the same XML document. The type xs:IDREFS represents a whitespace-separated list of one or more xs:IDREF values.

The schema-aware version of MXQuery supports the types xs:ID, xs:IDREF and xs:IDREF. The NamedToken class has been extended to contain three fields: ID, IDREF and IDREFs where the corresponding values – if available – should be stored. For identifying the values of ID, IDREF and IDREFs, two input adapters based on SAX (ValidatedSaxImportAdapter and NonSchemaValidatingSaxImportAdapter) can be used. As described above there are three ways in which an ID attribute can be identified. Respectively, the ValidatedSaxImportAdapter first checks for the xml:id attribute among the attributes of an element. If such an attribute is identified, its value is stored at the ID field of the NamedToken representing the element which contains the xml:id attribute. In a similar way the ID field is assigned the value of an attribute whose type was found to be xs:ID. ValidatedSaxImportAdapter also implements the DeclHandler interface for dealing with DTDs. When a DTD is processed and an attribute of type ID is identified, an entry is appended to a list (Vector) of strings. The entry is a string of the form “element name”#"attribute name". In this way, when the attributes of an element are being processed in the startElement() method of the SAX handler, the attribute name combined with the element it belongs to are checked against the above mentioned list of strings. If such an entry is found in the Vector, the value of the attribute is used as the value of the ID field of the NamedToken that represents the element containing the attribute.

A similar procedure is followed for identifying IDREF and IDREFs with the sole difference that there is no “default” attribute like xml:id indicating IDREF and IDREFs, Hence, these attributes are identified only through schema or DTD information about attribute types.

When the NonSchemaValidatingSAXImportAdapter is used, schema validation is not available. In this case the procedure is exactly the same with the difference that there is no information through schema validation.
3.14 Construction Modes (strip, preserve)

The construction mode governs the behavior of element and document node constructors. If construction mode is “preserve”, the type of a constructed element node is xs:anyType and all attribute and element nodes copied during node construction retain their original types. If construction mode is strip, the type of a constructed element node is xs:untyped; all element nodes copied during node construction receive the type xs:untyped, and all attribute nodes copied during node construction receive the type xs:untypedAtomic [6]

Construction mode “preserve” is now supported in the schema-aware version of MXQuery. The implementation details are briefly described below:

When a query is parsed, the iterators corresponding to the nodes that are copied, within a direct constructor are marked by setting the constrModePreserve field accordingly. At the XMLContent iterator, when the NamedTokens of the iterators are copied, if the Iterator whose tokens are being copied has been “marked”, the copyStrip() method of NamedToken is called. The copyStrip() method does not maintain the type information of the token copied. For computed constructors, the computeCompConstructor() and collectTextValues() methods of XMLContent have been modified and types are not maintained when construction mode is “strip”. The collectTextValues() method is used in order to create the content of an computed constructor. Adjacent text nodes or atomic values of a sequence are merged in a single text node. Other types of tokens (e.g. NameTokens) are simply forwarded to the output. At this point, a slight modification has been applied, and NamedTokens loose their type annotation (namely copyStrip() is used again) before being forwarded to the output, if the construction mode is set to “strip” In both cases of direct and computed constructors the types of the constructed elements (untyped or anytype) are assigned correctly depending on the construction mode. This is done during the parsing phase for direct constructors and again within the body of computeCompConstructor()- where the first iterator of a computed constructor is evaluated and a NamedToken representing the constructed element is created - for computed constructors.

3.15 Node Kind Tests implementation

Given the support for UDTs and XDM construction from the PSVI, the full list of node kind tests has been implemented. More specifically, ElementTest and NodeTest with type information have been added and SchemaElementTest and SchemaAttributeTest have been implemented in the schema-aware version of MXQuery. Two different integer encodings have been added in the Type System for the representation of the SchemaElement and SchemaAttribute tests (35 & 39 respectively). In general, sequence type matching has become type - (and user-defined type) aware. The integration of type awareness in sequence type matching was rather smooth and “clean-cut” by simply extending the logic of sequence type matching with type checks, trying to “match” types carried by TypeInfo objects with the type annotations of tokens in conformance with the rules for sequence type matching as specified in [6], Section 2.5.4.
3.16 Functions and Operators
The implementation of the XQuery functions and operators (see packages ch.ethz.mxquery.functions.*, ch.ethz.mxquery.operators.*) had to be revisited. Since the DataValuesIterator iterator lies in the “heart” of the system and most of the iterators implementing the functions and operators are dependent on it, integration of the updated version of DataValuesIterator was rather smooth. However, in some cases the iterators had to be modified in order to take UDTs into account. When a UDT is encountered its type is extracted and the built-in base type it has been derived from is identified. From that point on, the token carrying the UDT is treated as if it was annotated with the corresponding built-in base type.

3.17 Testing Phase - Unit tests
After the completion of the requirements, design and implementation phases for each of the features presented above, a testing phase took place consisting of standalone tests for the features that were updated and implemented.
MXQuery uses a Testing framework for checking correctness and compliance with the XQuery recommendation. The MXQuery test framework Test Suite consists mainly of the XQuery Uses Cases tests but is also contains a number of JUnit tests for the separate components of the MXQuery Engine.

The extended type system and the updated parser (implementing schema imports) were tested as a separate component of the engine. New JUnit tests were added to the Test Suite to check the methods of class Type and TypeDictionary (see ch.ethz.mxquery.model.types) that were updated/added to support UDTs.

JUnit tests checking the correctness of the implementation of schema imports were also added to the Test Suite with main focus on testing the correct association of modules to ISSDs (Type Dictionary) and whether the exceptional/erroneous cases are handled in conformance with the specification.
The results of the execution of the MXQuery Test Suite suggest that the integration of the extended Type System and schema-related features in the engine has been successful.
Chapter 4

User and developer guide

4.1 Execution modes and command line interface

With the addition of the schema-awareness features in MXQuery, the need for clearly separating different modes of execution for the engine is apparent, especially taking into account that these features introduce a significant performance overhead compared to the basic version of MXQuery that did not support schema, user-defined types and related functionality. The remainder of this section presents the different modes of execution that have been designed for MXQuery and also the way these modes can be imposed both at the API level and the command line interface level.

4.1.1 Modes of execution

4.1.1.1 Basic Mode

**Activation:**
This is the default execution mode.

**Description:**
All schema-related features are treated as "unsupported".

4.1.1.2 Schema Aware Mode

**Activation:**
This mode can be enabled either by calling Context.globalContext.enableSchemaAwareness() or by specifying the -sa option at the command line.

**Description:**
Schema imports, validate expressions, UDTs and UDT constructor functions are supported. Input is validated only if enclosed in a ValidateExpr
4.1.1.3 Validate Mode

**Activation:**
This mode can be enabled either by calling `Context.globalContext.enableValidation()` or by specifying the -val option at the command line. If set, also enables schema-awareness. Validation mode (strict|lax) can be optionally specified either by or by appending the keyword strict or lax within square brackets to the -val switch in the command line. (command line: -val [strict | lax])

**Description:**
Input is always validated. Validation mode (strict|lax) can be optionally specified. Default validation mode is "strict". If input is enclosed in a ValidateExpr, then the validation mode used, is the one specified (or implied for "strict") in the ValidateExpr.

4.1.1.4 Normal input Mode

**Activation:**
This is the default behavior for input validation

**Description:**
Input is validated only if enclosed in a ValidateExpr.

The small number of examples listed below present the different cases that can occur. For convenience an example of the command line arguments used to initialize the engine execution environment is given, along with the query to be executed and the expected behavior, error or result.

Note: The command line option -schemaFiles, which is not associated with any specific mode of execution as the ones described above, allows the user to specify the URI of schema files to be pre-loaded when the engine is initialized. The schema documents to be parsed should be provided as a list of semicolon separated strings. The URI of the last document in the list should also be followed by a semicolon.

* e.g. -schemaFiles file1.xsd;file2.xsd;
4.2 Examples

The files that are used in these examples (query1.xq, query2.xq, along with the schema document shapes.xsd and the input XML file, shapes.xml can be found in the Appendix, section A.2).

1. -sa -val -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.DynamicException: cvc-elt.1: Cannot find the declaration of element 'shapes:rectangles'.
```

2. -val -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.DynamicException: cvc-elt.1: Cannot find the declaration of element 'shapes:rectangles'.
```

3. -val [strict] -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.DynamicException: cvc-elt.1: Cannot find the declaration of element 'shapes:rectangles'.
```

4. -val [lax] -f query1.xq

Result:
```
true
```

5. -sa -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.TypeException: Sequence Type Matching Failed:
Wrong type: Expected {http://www.shapes.com}:rectanglesCollection, encountered type :xs:untyped
```

6. -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.StaticException: Schema Imports are not supported in this version
```

7. -val [lax] -f query1.xq

Result:
```
ch.ethz.mxquery.exceptions.DynamicException: cvc-elt.1: Cannot find the declaration of element 'shapes:rectangles'.
```

8. -val [strict] -f query3.xq

Result:
```
true
```
### 4.3 Execution Modes - Java API

Developers can use the MXQuery API to include query execution in their applications. This section describes how the MXQuery Java API can be used to impose the different modes of execution already described in 5.1.

Class `Context` (ch.ethz.mxquery.query.Context) provides the following methods for imposing the different modes of execution and validation presented above. (most likely they will be used at global context level : `Context.globalContext`, affecting the execution of the entire engine, when the engine is initialized.)

- `enableSchemaAwareness()` : Activates the schema-aware mode
- `enableValidation()` : Activates validation mode

Note: At the API level, schema awareness needs to be explicitly enabled.

- `setValidationMode(String mode)` : Used to set the mode of validation. Developers are expected to specify “strict” or “lax” as the value of the mode argument. Default validation mode is “strict”.

A complete example of using the MXQuery Java API for executing a query and setting execution and validation modes is listed below:

```java
Compiler compiler;
PreparedStatement statement;
String query = "...."; //the query to be executed
Context ctx = new Context();

ctx.globalContext.enableSchemaAwareness(); //enable schema-awareness
ctx.globalContext.enableValidation(); //enable validation
ctx.globalContext.setValidationMode("lax"); //input validation mode is set to"lax"

compiler = new CompilerImpl(); //instantiate the compiler
statement = compiler.compile(ctx, query); //compile query

Iterator result = statement.evaluate(); //evaluate iterator for query
XDMSerializer ip = new XDMSerializer();
OutputStream out = System.out;
ip.eventsToXML(out, result); //print result
```
Chapter 5

Stream Schema

5.1 Foundations
This section is a short introduction to streams, known problems and issues about stream processing and the motivation for stream schema.

5.1.1 Stream Definition
A stream is a series of items carrying information. The items are coming either from a storage or are freshly generated. Storage might be a database or simply a hard disk residing in any computer around the world. If the items are generated, the generating device might be some sensor measuring and reporting e.g. temperature in a room every second or a feed author writing e.g. articles for an RSS-Feed or blog. The series might be infinite. Generated streams like measurements from a weather sensor are usually infinite. Finite streams are usually coming from storage e.g. the result of a database query. Streams are accessed by long-during or even infinite and ad hoc queries. A long-during query often uses window functions or aggregates over all data items arriving from the moment it is asked. Therefore, every incoming item might trigger a new output or is simply integrated to the aggregate.

5.1.2 Known Issues in Data Stream Processing
Problems that arise when using streams mainly have to do with memory management and concurrency. In the scope of this thesis, the problems can be divided in three main categories; general streaming problems, particular language problems and (MX)Query implementation specifics.

A general problem of streams is that it is not possible for any type of stream processing system to materialize all items passing by during execution, since, if a stream is infinite, there is an infinite number of items and thus “infinite” memory needs. As far as concurrency is concerned, scheduling issues also arise. For the special case of MXQuery, the problem with concurrency is that if we select sequences from the stream to work on, there may be some infinite sequences that prevent all following ones from being seen or processed. Hence, parallel handling should somehow be enabled. An approach could be to pre-detect infinite sequences and delay them until the finite ones have been handled. Note that there may also be an infinite number of finite sequences. Some way is needed to describe information about streams, exploit this information to detect such problematic queries on streams and adapt the way of handling them or even abort the execution if there is no possible way of handling the query.

From the perspective of a query processor an additional issue is described below:
In the presence of infinite streams, it is obviously impossible to store all arriving items to be able to answer queries after the last item has arrived. Therefore, only aggregating queries that can be calculated cumulatively or the so-called rolling aggregates are possible to answer. This introduces a problem for blocking operators like “sort” which has to “consume” the whole input before being able to start producing results.

A similar more language specific problem can also show up when regarding predicate-based windows: It is not always possible to specify whether a condition will ever be met and therefore if a function will terminate.

### 5.2 The stream schema proposal

A recent proposal for a schema for streams [2] is an attempt to deal with the issues presented in the previous section. The novelty of this proposal lies in the fact that it is focused on the description of static stream properties – so far related work in this area was based on the description of dynamic characteristics of streams.(see [2], Chapter 9 for more details).

The proposal suggests a general solution for describing streams by extending the concept of regular expressions. The solution is also described formally for the special case of XML streams. In order to describe streams consisting of XML items, the same ideas are applied and an extension to XML Schema has been formally described.

An overview of the basic stream characteristics that the proposal suggests are given in this section by means of a simple graphical example. For a detailed presentation of the proposal please refer to [2].

#### 5.2.1 Graphical Introduction to Stream Schema

A simple sequence of items and the way our view of this stream changes when specific characteristics of this sequence of items are described, is shown in Figure 5.1.

Pattern identification, sequence repetition and interleaved sequences, stream combination and constraints about the relationship between item values are the properties that can be described about a stream. The figure is explained in more detail below:

Starting from the simple sequence of items, the first property that could be described, is whether the items of the stream follow a specific pattern. The pattern could also be described by extending the concept of regular expressions and reusing regular expression syntax. In the working example, (A|B)* could describe the pattern in the stream, where A & B refer to item types A and B and not necessarily characters as in the common case of regular expressions.

A stream can also contain several instances of the sequence (interleaved sequence) as shown in the third row of the figure. The proposal defines a key mechanism for the association of item to sequences. In the current example the id value of each item serves as a key providing the subsequence id. For the needs of visualization, items corresponding to different subsequences are marked with different colours accordingly.

Another group of stream characteristics that the proposal formally describes, have to do with relationships between values of different items of a stream. More specifically value constraints can be specified in schema, requiring that specific consecutive item values should satisfy a given relationship. In the working example, the constraint in question, is whether the timestamps of items grow monotonously. If the given stream contains interleaved sequences, the constraint is satisfied (see row 4 of the figure) whereas the same constraint is violated if the sequence of items is non-interleaved (3rd row). The use of keys can also specify to which items a given constraint is applicable. Thus, when keys
are used, the term “consecutive items” will refer to consecutive items matched by the key constrained. A last feature not shown in the figure, is the option of also describing set relationships in value constraints, meaning that one could require that item values are members of a given set.

Finally, a stream can also consist of a combination of streams. Again, the patterns that can be identified on each of the constituent streams, can be described by a regular expression and in general all the properties and rules presented in this section for a single stream can of course refer and be applied to each of the streams “constructing” the overall stream combination.
A sequence of items

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
</tbody>
</table>

Patterns

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
</tbody>
</table>

Regular Expression: \((A|B)^*\)

Interleaved sequences

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
</tr>
</tbody>
</table>

Regular Expression: \((A|B)^*\)

Value of id => Interleaved Subsequence

Next Value Constraints

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>

Regular Expression: \((A|B)^*\)

Constraint: \(t_{\text{next}} > t_{\text{current}}\) (Violated)

Next Value Constraints ( & interleaved)

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>

Regular Expression: \((A|B)^*\)

Value of id => Interleaved Subsequence

Constraint: \(t_{\text{next}} > t_{\text{current}}\)

Stream Combination

<table>
<thead>
<tr>
<th>ID</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>

Stream 1: A* Stream 2: B*

Figure 5.1
5.3 Changes in semantics and optimizations

Having a description of a stream and the validity information available a number of changes in the semantics of processing and a series of optimizations can be performed. As far as the semantic changes are concerned, a typical example is the change in the behavior of blocking operators such as the “sort” operator. This operator would normally need to consume the whole input before being able to produce results. However, if stream schema and stream validity information are available suggesting that the items of the stream “arrive” in an already sorted order, the behavior of the operator can change, and the items can be directly forwarded to the output as they arrive. In a similar way a “warn” or “abort” behavior can now be an option for blocking operators when the information about an infinite sequence is available. Especially when constraints that have been defined in schema prove not to be satisfied execution and the processing within an operator can be aborted already at an early stage.

As far as optimizations are concerned, these mainly have to do with memory management and decision making about the items that have to be materialized and memorized given the schema information, as well as possible window rewrites when windowed queries are used.

An example of a possible window rewrite is presented below:

A simple sequence of items related to events produced by a sensor (card reader) installed at the entrance of a building is presented in Figure 5.2. The “in” and “out” items correspond to entrance and exit events for a given person (a person is represented by an instance of a sequence). The simplest property that can be described about this sequence has to do with the pattern that each sequence instance follows. It can easily be seen on this figure that the pattern identified is the repetition of the IN – OUT sequence. Namely, each person has to leave the building before entering again. This information can be exploited in order to rewrite a windowed query on the card reader events and more specifically use sliding or tumbling windows instead of landmark. A visualization of the different window types for a similar scenario is given in Figure 5.3 for convenience.

![Figure 5.2 Entry gate events sequence](image)

For a detailed analysis of possible semantic changes in processing and optimizations please refer to [2].
As mentioned already in the previous sections, the general idea of the stream schema proposal has also been applied to an extension of XML Schema for the description of XML streams. The virtue and strong point of this extension is that it is really built “on top of” the basic schema specification. The main change is that the notion of “root node” has been replaced by that of a “root sequence” – which of course also covers the “root node” case, when a sequence consists of a single item and the root sequence actually “degenerates” into a root node. Two new schema components have been added for defining root sequences: streamChoice, for stream combination and streamSequence for the general root sequence case. Additional schema components have been formally described for the definition of other properties such as value constraints and keys. The stream schema extensions are presented in detail in [2]. The formal definition of the XML Schema components and the schema for streams schema are listed in the Appendix in their most recent version (after a few minor modifications that were made in the framework of this thesis).

A simple XML Schema for streams example is given in Figure 5.4.

A streamSequence declaration follows the xs:schema root element. The pattern “in-out” is described.

The streamInfo element contains information about the stream properties. The described pattern can be infinitely repeated and interleaving instances of this sequence is also supported. The streamConstraint element defines a key for the association of each item to a (sub)sequence by means of an XPath expression. The maximum number of sequence instances is specified by the value of the variety attribute. Finally an example of the way next value constraints are defined is shown (streamNext...
element). Again an XPath expression given as the value of the “selector” specifies which items will be affected by this constraint. In summary, the streamNext element shown here describes the constraint that the values of the “time” attribute of the in/out elements will be monotonously growing. The values to be compared are also described with an XPath expression.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:streamSequence>
    <xs:element name="in" type="inout"/>
    <xs:element name="out" type="inout"/>
    <xs:streamInfo repeating="infinite" interleaved="true">
      <xs:streamConstraint name="seqKey" type="key" variety="50">
        <xs:field xpath="/0person"/>
      </xs:streamConstraint>
      <xs:streamNext selector="/ */">
        <nxt:comparator type="greater">"</nxt:comparator>
        <nxt:next xpath="/0time"/>
        <nxt:current xpath="/0time"/>
      </xs:streamNext>
    </xs:streamInfo>
  </xs:streamSequence>
</xs:schema>
```

Figure 5.4 - A basic stream schema example
Chapter 6

Xerces for Streams

The main goal of the second part of this thesis has been the extension of an available tool for parsing and validation of XML data in order to support the XML Schema Extensions that have been described in the stream schema proposal (see [2]). More specifically, the widely used Xerces parser and validator had to be extended in order to support the following features: parsing of Stream schema documents, parsing of XML element streams and validation of an XML element stream against a given XML stream schema (validity checking, PSVI extension with stream-related information and access to this additional information).

6.1 Introduction to Xerces

Xerces is a family of software packages for parsing and manipulating XML. It is an open source Apache project. It provides both XML parsing, validation and generation and is a fully conformant XML Schema processor. The latest version of Xerces (Xerces2) introduced the Xerces Native Interface (XNI); a framework for communicating a "streaming" document information set and constructing generic parser configurations. Streaming information set means the document information that can be communicated by parsing the document in a serial manner. In other words, it is the information received as-you-see-it. An XNI parser provides this streaming info set to a registered document handler. The remainder of this section will provide a high-level overview and description of the architecture of the Xerces parser taken from the Xerces official project web site [2].

The Xerces Native Interface is used to implement the Xerces parser from a set of modular components in a standard configuration. This configuration is then used to drive the DOM and SAX parser implementations provided with Xerces2.

A parser written to conform to the Xerces Native Interface (XNI) framework is configured as a pipeline of parser components. The document's "streaming" information set flows through this pipeline of components to produce some sort of programming interface as the output. For example, the pipeline could produce a W3C Document Object Model (DOM) or a series of Simple API for XML (SAX) events.

The XNI parser pipeline is any combination of components that are either capable of producing XNI events, consuming XNI events, or both. All pipelines consist of a source, zero or more filters, and a target. The source is typically the XML scanner; common filters are DTD and XML Schema validators, a namespace binder, etc; and the target is the parser that consumes the XNI events and produces a
common programming interface such as DOM or SAX. The following diagram illustrates the basic pipeline configuration.

![Figure 6.1 Xerces basic pipeline configuration](image)

The Xerces2 parser, the reference implementation of XNI, contains more components than the basic pipeline configuration diagram shows. The following diagram shows the Xerces2 pipeline configuration. The arrow going from left to right on the top of the image represents the flow of document information and the arrows on the bottom of the image represent the DTD information flowing through the parser pipeline.

![Figure 6.2 - The Xerces2 reference implementation pipeline configuration](image)

As the diagram shows, the "Document Scanner" is the source for document information and the "DTD Scanner" is the source for DTD information. Both document and DTD information generated by the scanners flow into the "DTD Validator" where structure and content is validated according to the DTD grammar, if present. From here, the validated document information with possible augmentations such as default attribute values and attribute value normalization flows to the "Namespace Binder" which applies the namespace information to elements and attributes. The newly namespace-bound document information then flows to the "Schema Validator" for validation based on the XML Schema, if present. Finally, the document and DTD information flow to the "Parser" which generates a programming interface such as DOM or SAX.
The XNI parser configuration framework provides an easy and convenient way to construct different kinds of parser configurations. By separating the configuration from the API generation (in each specific parser object), different parser configurations can be used to build a DOM tree or emit SAX events without re-implementing the DOM or SAX code. The following diagram shows this separation. Notice how the document information flows through the pipeline in the parser configuration and then to the parser object which generates different APIs.

![Figure 6.3 – Xerces modular architecture](image)

6.2 The Xerces extension

The Xerces2 reference implementation was modified in order to support streams parsing and validation based on XML Stream Schema information. This section presents the basic implementation details for this extension.

6.2.1 The general approach

The Xerces reference implementation supports parsing and validation of XML Documents. The main idea behind the extension was that the single document that the existing implementation supports, could be seen as an individual item of a stream. Thus, for the needs of parsing and validation for each individual item, the existing infrastructure should be sufficient. Each item should be treated as a “black box”. The parser should be modified in order to “allow” more items to be parsed after the end of each item (previously document) is reached. As far as the validator is concerned, it can be seen as an additional layer that “observes” the properties of the “higher view” of the stream and makes use of this information in order to decide about the validity of the stream and augment the PSVI information of each item accordingly. This general approach behind the extension is visualized in Figure 6.4.
6.2.2. Affected components

Three main components of the Xerces2 implementation that had to be extended or modified for streams support: The parser of schema documents, the parser of XML Documents (streams) and the Validator. Details about the most interesting points of the modification of the above mentioned components are provided in the following subsections.

6.2.2.1 Schema documents parser

The schema documents parser is used to parse grammar documents and create a model for the representation of the properties of the several schema components. Xerces uses a kind of a global registry implemented by a number of hash tables for storing the information of the globally declared schema components (global element, attribute and type declarations) and allowing lookups based on the name of the global schema component.

The schema documents parser is implemented by a number of schema component traverses defined in the classes of package org.apache.xerces.impl.xs.traversers. Class XSDHandler is the general class used to coordinate the construction of a grammar object corresponding to schema. For the parsing of stream Schema documents a number two more traversers were written and added to the above mentioned package; XSDStreamChoiceTraverser and XSDStreamSequenceTraverser for handling streamChoice and streamSequence components respectively. These two new traversers follow the same style used in the traversers for parsing schema documents compliant with the basic XML Schema version.

For the representation of the new schema components a number of classes have been defined. When a component is parsed, the corresponding property values are extracted from the document and an instance of the class used to represent this component is created. When designing the classes for the model representation the general approach that was followed, was to be consistent with formal description of the schema components as it has been provided in [2], Chapter 5. The classes for the model representation have been added to packages org.apache.xerces.xs and org.apache.xerces.impl.xs.
in the case of components bound to the “xs namespace-http://www.w3.org/2001/XMLSchema“. Components belonging to the “nxt namespace-http://www.dbis.ethz.ch/streamingXQueryP-nxt“ have been added to package “org.apache.xerces.nxt“.

For example, the properties of the of the streamInfo XML Schema component are described by the proposal as follows:

**Schema Component: StreamInfo Definition**

- \{annotations\}   A sequence of Annotation components
- \{repeating\}      The number of iterations, unbounded or infinite
- \{interleaved\}    A boolean.
- \{streamNext\}     A set of streamNext Definition Schema Components.
- \{streamConstraint\} A set of streamConstraint Definition Schema Components.

Class XSStreamInfoDecl (org.apache.xerces.impl.XSStreamInfoDecl) is designed to represent all these properties with fields corresponding with a one-to-one relationship to the properties listed above. More specifically the class contains the fields:

```java
  int fRepeating = 0;
  boolean fInterleaved = false;
  List<XSStrnextNextDecl> fStreamNext;
  List<XSStrstreamConstraintDecl>  fStreamConstraint;
  XSObjectList fAnnotations;
```

In addition to the fields, methods are defined in all classes used for the model representation, serving as accessors to the components properties information.

Although a formal description of the API for accessing the new introduced schema ocmponents, has not been provided in the framework of this thesis, the definition of the model for the representation of the XML Stream Schema components can be regarded as a first step towards the specification of an extension to the XML Schema API, for streams.

Upon completion of the traversal of the components and the parsing of the schema documents, an object representing either a streamSequence or streamChoice component declared in schema is added to the global components registry (see fGlobalStreamModelGroup field of class SchemaGrammar (org.apache.impl.xs.SchemaGrammar)
6.2.2.2 XML scanner

For stream parsing to be made possible, the scanner of the reference implementation had to be modified.

The scanner consists of a number of classes (see class names containing “ScannerImpl” in package org.apache.xerces.impl.xs) which cooperate for scanning fragments of the input XML document. From a high-level perspective, the scanner could be seen as a state machine. Upon parsing a sequence of characters from the input, the state machine state is updated. Transitions to illegal states or specific selected states indicate that the document is not well formed. The scanner also cooperates with the Validator, sending events that the validator is supposed to handle and this way validate the content and structure; augment the InfoSet, and notify the parser of the information resulting from the validation process.

The extension of the scanner required a very small modification of the state machine, namely a new state had to be introduced (please see `SCANNER_STATE_CHECK_SEQUENCE`, defined in class XMLDocumentFragmentScannerImpl ). This state is reached when the end of a document would be reached in the basic version of the parser. It is actually an intermediate state, indicating that the scanner is expecting either the end of the input file or a new item. When the end of file is identified, the scanner changes to the corresponding state, as it would do before. If more input is identified, instead of raising an error, as this should be illegal in the basic version of the parser, a “second chance” is given, and the scanner is reset, by changing to the state indicating that the beginning of a new document is expected (a new item).

**Note:** Xerces supports a features and properties mechanism for the definition of its behaviour and its parametrization. For a clear separation between the extended “streaming” version and the basic version a new feature has been added. All the functionality presented here and in the following subsection about the validator is supported only when the “http://apache.org/xml/features/streamingMode” feature is enabled.

A sample program that uses the extended version of Xerces for the validation of a sequence of XML items can be found in the Appendix, section A.4.

6.2.2.3 Validation

The Xerces2 validator is implemented by class org.apache.xerces.impl.xs.XMLSchemaValidator.

This class had to be modified in order to support the stream schema extension in a way that behavior of this component would remain unchanged for the basic version and the already existing functionality could be reused to the greatest possible extend.

The main features that had to be supported were per item validation, regular expressions support, interleaved sequences and item with subsequence association and next value constraints. More details about the way each of the features has been supported are given in this section. All these features and the corresponding schema properties that they check, are activated when the http://apache.org/xml/features/streamingMode feature is enabled.
**Pet item validation**

The solution to the problem of “per item validation” is based on the observation that each item can be regarded as what would be a single document in the basic Xerces version and treated as a black box with respect to validation. The basic schema validator can be used to validate each individual item exactly in the same way it is used to validate a single document. Since the scanner has been modified to allow parsing a sequence of items, every time a new is encountered, the state of the validator simply needs be reset. (see method resetValidatorState() of XMLSchemaValidator).

**Regular expressions support**

Regular expressions support refers to identifying patterns – which have been described in the body of streamChoice of streamSequence elements- on a stream. The observation was made that this actually constitutes exactly the same problem with checking complex type content. The basic version of Xerces already provides a component for dealing with that problem. Xerces creates a Deterministic Finite Automaton (DFA) based on regular expressions (complex content) defined in schema (see method getContentModel() of class XSComplexTypeDecl). Each time a new element is encountered during parsing, a check is performed about whether this element is allowed as input based on the current automaton state (if the corresponding transition to a new state for this input and given state exists). The same functionality has been added to the class representing streamSequence declarations (see class XSSStreamSequenceDecl, getContentModel()). For regular expressions support, a DFA is created but now based on a streamSequence declaration. Transitions are now triggered and checked at the stream level, namely every time a new item is encountered. The validity of this individual item is checked first as described in the previous subsection on per item validation. Once this check is performed, a global level check is also performed by using the global DFA used for the whole stream. Hence, the DFA for the internal content of each item is reset several times (once for each individual item), where the DFA for the whole stream is only initialized in the very beginning of the validation process and never reset.

**Interleaved sequences and association of items with sequences**

The problem of associating items with sequences actually consists of two sub-problems. The first problem is that of applying a key constraint to each item and extracting the key value for the item and the second involves maintaining a “book keeping” mechanism, for storing all the different key values and having a way of mapping given values to subsequence ids. The former has been resolved by reusing and modifying the infrastructure Xerces provides for dealing with identity constraints in the basic schema version. (see classes Field and XPathMatcher, package org.apache.xerces.impl.identity) XPathMatcher has been extended by class ConstraintKeyXPathMatcher. The main modification has to do with the fact that the value of the matched key is stored and can be retrieved. A concatenation of values is also supported in order to deal with composite keys, if a number of XPath expressions have been given for a single constraint. A check is also performed to make sure that all nodes matched by the same XPath expression for the same item have a common value. Otherwise an error is raised.

For the book keeping mechanism, a global Hashable mapping values to ids was added to XMLSchemaValidator. Each time a key value is extracted, a lookup is performed in the Hashtabla. If the lookup is successful, the corresponding id is returned. Otherwise, a new entry is inserted having the value in question as key, and the overall size of the table incremented by one as the value (Hashtable
value for this given key). This way new sequence ids are created when the first item of subsequence appears. Before a new entry is made, the size of the Hashtable is compared with the value of the variety property of the streamInfo component, to check whether the allowed number of interleaved sequences has been exceeded. An error is raised if necessary.

**A brief note on errors:**

For schema related errors new error codes and descriptions have been added to the “errors properties file” that Xerces uses to define error codes and their descriptions (see org.apache.xerces.impl.msg, file XMLSchemaMessages.properties). The general approach that was taken is, that an error is always raised when validity is not satisfied. Execution and processing will however continue and it is up to the user application to handle this error. The user application, that actually drives the parsing and validation process, can for example explicitly stop the execution of the parser.

**Next value constraints check**

Next value constraints have been implemented only for non-interleaved sequences and for constraints that do not contain operators definitions. For next value constraints implementation a number of problems needs to be solved; XPath matching to select the items to which a given constraint is applicable and the design of a mechanism for checking the constraints themselves after extracting and storing the value information from the affected items.

The requirements of XPath matching are slightly different than those of the interleaved sequence case. The selector XPath expression is not subject to the constraints about common key values and in general there is no interest in the actual values of the matched fields; the information about whether a specific node exists in an item and is matched by an expression is sufficient. Thus, the already existing XPathMatcher class was reused.

Once an item is selected for checking, a number of issues need to be resolved. The value(s) of the item to be checked have to be extracted. Values are again specified in terms of an XPath expression. This XPath expression matching is again different than the key constraint case but also the general XPath matching case, in that all the values of nodes that are matched have to be somehow stored and no restrictions apply to them.

Class NextRuleResolver (package org.apache.impl.xs) is used to perform all this actions related to constraint checking. It contains one NextRuleXPathMatcher (a subclass of XPathMatcher) for each type of element defined in a next rule declaration: first, next or current. NextRuleXPathMatcher stores the matched values in a list, so that each one of them can be taken into account in a constraint check. NextRuleResolver also has its own lists in order to store the current and next values that are provided by the matcher. The general procedure could be described as follows: When a new item appears, values are extracted by applying the XPath expressions and checked with the already stored ones. After the check has been performed, the values of the new item become the current. The “first” item value constraint is checked when the very first (matched) item of the sequence is encountered and from then on every time the sequence is repeated (again for the first item of the repeating sequence).
Chapter 7

Results of the implementation of the XML Stream Schema proposal

7.1 Supported features overview
The extension of Xerces for the support of a subset of the features described in the XML Stream Schema proposal has been successful. The extended version of Xerces can handle sequences of items, perform per item validation based on the basic XML Schema version and also validate the sequences against XML Stream Schema documents.

At the time of writing (July 2008) the list of supported features is the following:

- Regular expressions support (only for streams described by a streamSequence component).
- Repeating and Interleaved Sequences
- Keys for the association of items to interleaved subsequences (streamConstraint)
- Next Constraints (only for value (first, next and current) comparisons, without operators support and only for non-interleaved sequences)

7.2 Limitations
The process of extending Xerces in order to handle streams and implement the XML Stream Schema proposal also served as a “feedback loop” for the Stream Schema Proposal itself in terms of identifying issues in the feasibility of implementation and applying a few refinements or minor modifications to the formal description of the XML Schema extension.

Limitations and implementability issues have been identified in an attempt to support validation for interleaved sequences with a potentially large or infinite number of interleaved sequences. The main problems lie in the area of state management and item key to sequence id association. More specifically, the former case has to do with the fact that next value constraint checking requires storing the value of the most recently seen item for each one of the interleaved sequences (in the worst case). In the presence of an infinite number of interleaved sequences constraint checking might not be feasible to implement. Under a combination of conditions about the stream characteristics it might of course be possible, to perform a number of optimizations and store information about a subset of all the states (e.g. if known that now more items of specific subsequences are expected to appear). However, a formal description of the rules that should be satisfied in order to achieve this and in general the issue of deciding about whether value constraints are implementable for a given sequence are problems not
trivial to solve. The latter case, namely the problem of associating key values to sequence ids is also a very simple problem that reveals the limitations for infinite interleaved sequences support.

Figure 7.1 is an illustration of the problem. A streamConstraint element defining a composite key for item to subsequence matching is shown along with a sample item on which this constraint should be applied. The value of the composite key will for the item in question will be the string “cc” as shown in the diagram. In order to associate the item with a sequence, some sort of data structure will have to be used to store the values of keys, assign sequence ids to items and also ascertain that the maximum number of allowed interleaved sequences is not exceeded. The figure shows the simplest case, where a single linked list is used. It is clear, that in the worst case scenario, where the number of subsequences – and thus key values and nodes in the list- grows to infinity and no additional information can be exploited to optimize and delete unnecessary items from the list, the system will run out of memory.

A list of all the possible combinations of the stream qualifiers (repeating, interleaved, variety) and the “complexity” of the resulting overall stream is shown in the following table:
Table 7.2 Summary of stream qualifiers combination and resulting finite/infinite stream

<table>
<thead>
<tr>
<th>stream pattern qualifiers</th>
<th>resulting overall stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeating</td>
<td>interleaved</td>
</tr>
<tr>
<td>infinite</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>true</td>
</tr>
<tr>
<td>some number / unbounded</td>
<td>false (default)</td>
</tr>
<tr>
<td>some number / unbounded</td>
<td>true</td>
</tr>
<tr>
<td>0 (default)</td>
<td>false (default)</td>
</tr>
</tbody>
</table>

The cases where a stream does not contain interleaved sequences are not problematic. Even if sequences are repeated infinitely, there is no need to store any information apart from the value (or a bounded number of values) related to last seen item. When a new item arrives, the corresponding value(s) of the new item are extracted, the constraint is checked by comparing the new value(s) with the stored ones and the stored values are at the end replaced by the new. After the constraint check, information about the previous item values does not need to be stored any more.

The critical cases are the ones marked with the red rectangles in table 7.2. When interleaved sequences as supported, if the number of different sequence instances is infinite, or finite but large implementability issues arise as described above.
Chapter 8

Future Work

This chapter summarizes the future work based on the foundations that both parts of this thesis – XML Schema Support in MXQuery and Stream Schema Support- provided.

8.1 MXQuery

As far as MXQuery is concerned and given the fact that schema-awareness is currently supported in the engine and type information is available, the next step towards feature completeness could be Static Typing support.

Regarding the evaluation of the implementation of the schema-related features, a number of performance experiments could be performed. Thence, strong and weak points of the current design and implementation can be revealed. Especially the cost that one has to pay for validation (by using Xerces) could be measured in this way.

In case the results of the experiments suggest that the selection of Xerces for the “heavy” task of validation is a problematic design decision, the options of either building a dedicated validator tool or alternatively trying to evaluate and use a future more stable version of XMLBeans or some other tool that can be integrated more naturally in the MXQuery engine and is proved to perform better- could also be considered.

8.2 Stream Schema

On the part of Stream Schema support, a longer list of things still remain to be done.

Feature completeness, meaning the support of the XML Schema extensions in its entirety should the first goal. In addition to this, performance experiments to measure the overhead that validation for streams imposes would also be very interesting. Support of the general solution that the stream schema proposal describes for non-XML streams should also be on the agenda. Xerces and the XNI framework which can also be used to build parser configurations for any type of data (not only XML) could also be considered as a strong candidate framework for building a tool for parsing and validating streams of any type.

The stream schema support part could also be seen as a first step towards a formal proposal for an extensions to the PSVI specification, once the necessary information with which one can annotate stream items during the validation process is clearly specified. Similarly, the extension of the model that Xerces uses to represent the XML Schema components, and the methods that have already been
defined for accessing stream schema information, can be regarded as the foundation of a formally
described XML Schema API extension for streams.

Integration into different processing models completes the list of future work tasks. In this way the
impact of the Stream Schema proposal on different models can be examined in practice. At the same
time, the ideas for possible changes in semantics and optimizations can be made more concrete when
applied to a specific processing model and implementation.
Appendix

A.1 MXQuery Type System

Status

Type System:
- The Type system is a separate component
- Internal representation of the type is a 32 bit integer number.
- Translation between type internal representation and type “QName” (or vice versa) is implemented
- Lazy typing (implicitly assigned types):
  - Tokens with values surrounded by quotes or double quotes e.g. ('12’ or “aaa”) are assigned type xs:string
  - Integer value tokens (e.g. 7) are assigned type xs:integer
  - Real number value tokens (e.g. 5.5) are assigned type xs:decimal
  - Attribute tokens, element content tokens, etc. are assigned type xs:untypedAtomic with annotation to be attribute, element, text node kind
- Philosophy that a “not supported type” error is raised or the correct result is generated is completely implemented
- Functions/Iterators have static return type method (used for optimizations e.g. atomization is not needed). If the implementor does not take a decision, type xs:anyType is returned.
- There is no Query Rewriter for type based optimizations

Not Supported Atomic Types:
- xs:NOTATION, xs:NMTOKEN, xs:NMTOKENS, xs:ENTITY

Constructor functions for atomic types implemented. These types are supported in the appropriate iterators and functions.

Supported (implemented) Node Kinds:

<table>
<thead>
<tr>
<th>Name</th>
<th>Direct Constructor</th>
<th>Computed Constructor</th>
<th>Node Kind Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
<td>-</td>
<td>not supported</td>
<td>supported</td>
</tr>
<tr>
<td>element</td>
<td>supported</td>
<td>supported</td>
<td>element name test supported; type name test supported</td>
</tr>
<tr>
<td>attribute</td>
<td>-</td>
<td>supported</td>
<td>attribute name test supported; type name test supported</td>
</tr>
<tr>
<td>processing instruction</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>comment</td>
<td>supported</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>text</td>
<td>-</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>node</td>
<td>-</td>
<td>-</td>
<td>supported</td>
</tr>
<tr>
<td>item</td>
<td>-</td>
<td>-</td>
<td>supported</td>
</tr>
<tr>
<td>schema element</td>
<td>-</td>
<td>-</td>
<td>supported</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>supported</td>
</tr>
</tbody>
</table>

Note. *Schema element* and *Schema attribute* node kind tests have been added.

Now Supported:
- User defined (complex, union) types
- Typed XML, schema imports

Type operators:

<table>
<thead>
<tr>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>castable as</td>
<td>implemented;</td>
</tr>
<tr>
<td>cast as</td>
<td>implemented;</td>
</tr>
<tr>
<td>typeswitch</td>
<td>implemented;</td>
</tr>
<tr>
<td>instance of</td>
<td>implemented;</td>
</tr>
<tr>
<td>treat as</td>
<td>not supported; performs down-casting at compile time; it would make sense for static type checking;</td>
</tr>
</tbody>
</table>

Element and attribute content:

<table>
<thead>
<tr>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDATA section</td>
<td>implemented;</td>
</tr>
<tr>
<td>built-in character entities</td>
<td>implemented; (&lt; &gt; &amp; &quot; ' );</td>
</tr>
<tr>
<td>numerical character entities</td>
<td>not supported yet;</td>
</tr>
</tbody>
</table>

Type system at compile time:
- Optimizations based on return type of the function / operator:
  - Optimizations are applied in the constructor of relevant operator, not in a generic case
  - Atomization: If return type of the function is ComputableValue (internal type corresponding to xdt:anyAtomicType) the atomization is omitted. E.g. fn:avg(fn:sum(…)) where fn:sum return type is ComputableValue
  - BooleanIterator in some cases is omitted, because return type of the ComparisonIterator is xs:boolean. E.g. 3 eq 2 (special case when result of the comparison is empty sequence)
  - If return type of the expression in angle brackets is of type xs:integer, positional predicate iterator is created.

Type system at runtime:
- have tokens that implement values of types
- have constructors, functions and operators that deal with the values of types
- Implicit type conversions (currently built into the operators) E.g. expression 2.5 + 1: no explicit casting during compile time to cast token with value 1 of type xs:integer to token of type xs:double is added, while building operator tree. Value conversion is handled inside additive iterator. The type of the result token is deduced using type promotion and subtype substitution rules (defined in Type component).

Notes:
1) Path expression parsing and building of the iterator tree has been reimplemented.

1. Implementation of Type Encoding
1.1 Idea

To encode XQuery type hierarchy integer numbers are used. Integer number in Java (and Java CLDC) uses 4 bytes (32 bits) of memory. Each XQuery type is represented in a combination of bits. All types derived from that type have the same part as the parent type plus their own bits. In this way it’s easy to check if the type is subtype of another type.

E.g. xs:integer type is represented by number 12815 (bit representation: 11001000001111) and xs:anyAtomic type is represented by number 15 (bit representation: 1111). Since xs:integer type includes xs:anyAtomic type bits (at the end) it’s easy to find out that type xs:integer is derived from xs:anyAtomic type.

NB! All built-in types from the same level and the same branch in the type hierarchy tree should have the same encoding length (e.g. all primitive 20 atomic types will have bit 1 at position 10).

1.2 Reserved bits

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDT</td>
<td>0</td>
<td>0</td>
<td>NK</td>
<td>E/S</td>
<td>NK</td>
<td>NK</td>
<td>NK</td>
<td>NIL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

31st UDT: User defined type bit (1 – user defined type, 0 – built-in type)
27th E/S: bit for compatibility with existing event types (1 – START, 0 - END) only for element and document nodes
28-24 NK: Node kind bits (00000 – no node, 11000 element node/START_TAG, 00100 – attribute node/AttributeTest, 00010 – text node)
23: NILLED property

These checks are done very often. To simplify process of node kind determination, each of the node kinds has a separate flag.

Note: 32nd bit would be available if unsigned integers are used
### 1.3 Built-in type representations

<table>
<thead>
<tr>
<th>xs:anyType</th>
<th>Integer value: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xs:simpleType</th>
<th>Integer value: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xs:untyped</th>
<th>Integer value: 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xs:anyAtomicType</th>
<th>Note: 10(^{th}) bit equals 1 for all it’s subtypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value: 15</td>
<td></td>
</tr>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>C0</td>
<td>0</td>
</tr>
</tbody>
</table>

Atomic types encoded between 5\(^{th}\) and 10\(^{th}\) bits. 6 bits are used for each encoding in such a way 100000, 100001, 100010, 100100, etc.

<table>
<thead>
<tr>
<th>number</th>
<th>(internal type, subtype of xs:anyAtomicType)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note: 13(^{th}) bit equals 1 for all it’s subtypes</td>
<td></td>
</tr>
<tr>
<td>Integer value: 527</td>
<td></td>
</tr>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xs:decimal</th>
<th>Integer value: 4623</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary value:</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**xs:integer**
Integer value: 12815  
Binary value:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Subtypes of `xs:integer` should be added here

Integer has 3 subtypes, so we reserve 2 bits for the encoding and 1 bit in front with value 1 for hierarchy separation

**xs:nonPositiveInteger**
Integer value: 78351  
Binary value:

|   | 31 | ... | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtype hierarchy of `xs:NonPositiveInteger` started

**xs:negativeInteger**
Integer value: 209423  
Binary value:

|   | 31 | ... | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtype hierarchy of `xs:NonPositiveInteger` ended
**xs:long**  
Integer Value: 111119  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subtype hierarchy of xs:long started  

**xs:int**  
Integer Value: 242191  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**xs:short**  
Integer Value: 504335  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

**xs:byte**  
Integer Value: 1028623  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Subtype hierarchy of xs:long ended  

**xs:nonNegativeInteger**  
Integer Value: 94735  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Subtype hierarchy of xs:nonNegativeInteger started
xs:positiveInteger
Integer Value: 356879

```
| 31 | ... | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

xs:unsignedLong
Integer Value: 487951

```
| 31 | ... | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

xs:unsignedInt
Integer Value: 1012239

```
| 31 | ... | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

xs:unsignedShort
Integer Value: 2060815

```
| .. | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

xs:unsignedByte
Integer Value: 4157967

```
| 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Subtype hierarchy of xs:nonNegativeInteger ended

xs:double
Integer value: 5647
Binary value:

```
<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

xs:float
Integer value: 6671
Binary value:

```
<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

xs:string
Integer value: 543
Binary value:

```
<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

Start subtypes of xs:string
**xs:ID**
Integer value: 613919
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:IDREF**
Integer value: 744991
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

**xs:IDREFS**
Integer value: 1793567
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:ENTITY**
Integer value: 876063
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>...</td>
</tr>
</tbody>
</table>

**xs:ENTITIES**
Integer value: 1924639
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

End subtypes of xs:string

**xs:date**
Integer value: 559
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:time**
Integer value: 591
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**xs:dateTime**
- Integer value: 655
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:duration**
- Note: 12th bit equals 1 for all its subtypes
- Integer value: 783
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xdt:yearMonthDuration**
- Integer value: 2831
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xdt:dayTimeDuration**
- Integer value: 3855
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:boolean**
- Integer value: 575
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:QName**
- Integer value: 607
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:untypedAtomic**
- Integer value: 623
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>…</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**xs:anyURI**

- Integer value: 687
- Binary value:

| 31 | ... | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1 |
|----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0   | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  |

**xs:base64Binary**

- Integer value: 719
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:hexBinary**

- Integer value: 847
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:NOTATION**

- Integer value: 911
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:gDay**

- Integer value: 671
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:gMonth**

- Integer value: 799
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**xs:gYear**

- Integer value: 815
- Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
1.3 User defined type representation

User defined types (UDTs) are now supported. To denote user defined type 31st bit is reserved (value of this bit should be equal to 1). Bits 22 – 1 provide the index of the position of the type in the global dictionary of types.

Note: Bit position rules for the representation of subtype hierarchies do not hold for the case of UDTs.

1.4 Node kinds and special events

To support token based engine some of the event types remain.

**item**

Integer value: 0

**node**

Integer value: 2

**element** END_TAG

Note: In token based engine we have to keep separate types for start and end tag tokens!

Integer value: 34
**Element START_TAG**  
Integer value: 201326592 (28\textsuperscript{th} + 27\textsuperscript{th} bit is 1; see section 1.2 Reserved bits)  
Binary value:

| 31 | 30 | 29 | 28 | 27 | ... | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|----|----|----|----|----|-----|----|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 1  | 1  | 1  | 0   | 0  | 0 | 0 | 0 | 0 | 1  | 0  | 0  | 0  | 1  | 0 |

**Document END_DOCUMENT**  
Note: In token based engine we have to keep separate types for start and end document tokens!  
Integer value: 46  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Document START_DOCUMENT**  
Integer value: 67108906 (END_DOCUMENT + 27\textsuperscript{th} bit is 1; see section 1.2 Reserved bits)

**END_SEQUENCE event**  
Integer value: 64  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**START_SEQUENCE event**  
Integer value: 67108928  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>27</th>
<th>...</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Namespace (not supported)**  
Integer value: 50  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Processing-instruction**  
Integer value: 54  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Comment**  
Integer value: 58  
Binary value:

<table>
<thead>
<tr>
<th>31</th>
<th>...</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
attribute
Only annotation to the value type added (see section 1.2 Reserved bits NK - 26th bit)

text
Only annotation to the value type added (see section 1.2 Reserved bits NK)

To define the node kind test type the following constants are used:

    public static final int TYPE_NK_ITEM_TEST = 0;
    public static final int TYPE_NK_ANY_NODE_TEST = 2;
    public static final int TYPE_NK_PI_TEST = 54;
    public static final int TYPE_NK_COMMENT_TEST = 58;
    public static final int TYPE_NK_SCHEMA_ATTR_TEST = 39;
    public static final int TYPE_NK_SCHEMA_ELEM_TEST = 35;
    public static final int TYPE_NK_DOC_TEST = 46;
    public static final int TYPE_NK_TEXT_TEST = 42;
    public static final int TYPE_NK_EMPTY_SEQ_TEST = 64;

Element node kind tests and attribute node kind tests are represented in the same way as START_TAG and attribute nodes respectively. see section 1.2 Reserved bits NK

The parser uses these type constants while creating TypeInfo objects. TypeInfo object is used in node kind tests and sequence type verification.
A.2 Sample Files

shapes.xml

<?xml version="1.0" encoding="UTF-8"?>
<shapes:rectangles
xmlns="http://www.shapes.com"
xmlns:geom = "http://www.mygeomelements.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:type="shapes:rectanglesCollection"
xmlns:shapes="http://www.shapes.com">
  <rectangle id = "1">
    <origin>
      <geom:xcoord>0</geom:xcoord>
      <geom:ycoord>0</geom:ycoord>
    </origin>
    <width>5</width>
    <height>5</height>
  </rectangle>
  <rectangle id = "2">
    <origin>
      <geom:xcoord>2</geom:xcoord>
      <geom:ycoord>3</geom:ycoord>
    </origin>
    <width>6</width>
    <height>5</height>
  </rectangle>
</shapes:rectangles>

shapes.xsd

<?xml version="1.0" encoding="US-ASCII"?>
<xs:schema xmlns:my2="http://www.mygeomelements.com"
xmlns:my1="http://www.shapes.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.shapes.com"
elementFormDefault="qualified">
  <xs:import namespace="http://www.mygeomelements.com"
schemaLocation="/schema2.xsd"/>
  <xs:complexType name="rectanglesCollection" mixed="true">
    <xs:sequence>
      <xs:element name="rectangle" type="my1:recEntry" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="recEntry" mixed="true">
    <xs:sequence>
      <xs:element name="origin" type="my2:point" minOccurs="1" maxOccurs="1"/>
      <xs:element name="width" type="xs:positiveInteger" minOccurs="1" maxOccurs="1"/>
      <xs:element name="height" type="xs:positiveInteger" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
<xs:attribute name="id" type="xs:positiveInteger"/>
</xs:complexType>

<xs:simpleType name="myInteger">
  <xs:list itemType="xs:string"/>
</xs:simpleType>

<xs:simpleType name="myInteger2">
  <xs:restriction base="xs:integer"/>
</xs:simpleType>
</xs:schema>

query1.xq

import schema namespace shapes = "http://www.shapes.com" at "shapes.xsd";

let $r := fn:doc("shapes.xml")
let $a as element(*,shapes:rectanglesCollection) := $r/shapes:rectangles
return fn:data($a/shapes:rectangle[1]/shapes:width) instance of xs:integer

query2.xq

import schema namespace shapes = "http://www.shapes.com" at "shapes.xsd";

let $r := validate lax { fn:doc("shapes.xml") }
let $a as element(*,shapes:rectanglesCollection) := $r/shapes:rectangles
return fn:data($a/shapes:rectangle[1]/shapes:width) instance of xs:integer
A.3 XML Schema for Streams - Formal Description

A3.1. XML Representation of Schemas

XML Representation Summary: schema Element Information Item
<schema
attributeFormDefault = (qualified | unqualified) : unqualified
...
{any attributes with non-schema namespace . . .}>}

Content: ((include | import | redefine | annotation)*,
(defaultOpenContent, annotation*)?, (streamChoice
| streamSequence)?, ((simpleType | complexType
| group | attributeGroup) | element | attribute | notation),
annotation*)*) </schema>

XML Representation Summary: streamChoice and streamSequence
Element Information Item
<streamChoice
id = ID
{any attributes with non-schema namespace . . .}>}
Content: (annotation?, (streamSequence | element
| group | choice | sequence | any)*)
</streamChoice>
<streamSequence
id = ID
{any attributes with non-schema namespace . . .}>}
Content: (annotation?, streamInfo?,
(element | group | choice | sequence | any)*)
</streamSequence>

XML Representation Summary: streamInfo Element Information Item
<streamInfo
id = ID
repeating = (nonNegativeInteger | unbounded | infinite) : 0
interleaved = boolean : false
{any attributes with non-schema namespace . . .}>
Content: (annotation?, streamNext*, streamConstraint*)
</streamInfo>
**XML Representation Summary: streamConstraint Element Information Item**

```xml
<streamConstraint
  id = ID
  name = NCName
  type = (key) Required.
  variety = (nonNegativeInteger | infinite) : infinite Optional.
  xpathDefaultNamespace = (anyURI | (##defaultNamespace | ##targetNamespace | ##local))
><any attributes with non-schema namespace . . .>
  Content: (annotation?, field+)
</streamConstraint>
</streamConstraint>
```

**XML Representation Summary: streamNext Element Information Item**

```xml
<streamNext
  id = ID
  selector = a subset of XPath expression. Starts with "/. /"
  Can contain node tests at its end.
><any attributes with non-schema namespace . . .>
  Content: (annotation?, (nxt:comparator | nxt:dependency)+)
</streamNext>

<nxt:comparator
  id = ID
  type = (greater | greaterequal | equal | smallerequal | smaller) Required.
><any attributes with non-schema namespace . . .>
  Content: (annotation?, (nxt:first | nxt:next | nxt:current | nxt:value | nxt:operator){2})
</nxt:comparator>

<nxt:operator
  id = ID
  type = (addition | substraction | multiplication | division) Required.
><any attributes with non-schema namespace . . .>
  Content: (annotation?, (nxt:first | nxt:next | nxt:current | nxt:value | nxt:operator){2})
</nxt:operator>

<nxt:first
  id = ID
  xpath = a subset of XPath expression. Starts with "/. /".
```
A.3.2 Schema for schemas

schema

`<xs:element name="schema" id="schema">
...
<x:complexType>
  <xs:complexContent>
    <xs:extension base="xs:openAttrs">
      <xs:sequence>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
        ...
        </xs:choice>
        <xs:group ref="xs:streamModelGroup" minOccurs="0" maxOccurs="unbounded">
        ...
        </xs:group>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
...
</xs:element>

stream model group

`<xs:group name="streamModelGroup">
  <xs:annotation>
    <xs:documentation> This group is for the stream model groups which occur only at the top level of schemas. They are used to describe possibly infinite streams. </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="xs:streamSequence" type="xs:streamSequenceType"/>
    <xs:element name="xs:streamChoice" type="xs:streamChoiceType"/>
    ...
  </xs:choice>
</xs:group>
SchemaForSchemas streamInfo

```xml
<xs:complexType name="streamInfoType">
  <xs:attributeGroup ref="xs:defRef"/>
  <xs:sequence>
    <xs:element ref="xs:annotation" minOccurs="0"/>
    <xs:element ref="xs:streamConstraint" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="xs:streamNext" minOccurs="0" maxOccurs="unbounded"/>
    <xs:attribute name="interleaved" type="xs:boolean" default="false" use="optional"/>
    <xs:attribute name="repeating" default="0" use="optional"/>
  </xs:sequence>
</xs:complexType>
```

SchemaForSchemas stream constraint

```xml
<xs:element name="streamConstraint">
  <xs:annotation>
    <xs:documentation>
The stream-constraints to restrict stream model groups.
</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xs:annotated">
        <xs:element ref="xs:field" maxOccurs="unbounded"/>
        <xs:attribute name="name" type="xs:NCName" use="required"/>
        <xs:attribute name="type" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
              <xs:enumeration value="unbounded"/>
              <xs:enumeration value="infinite"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```
<xs:simpleType>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="infinite"/>
    </xs:restriction>
  </xs:restriction>
</xs:simpleType>
</xs:union>
</xs:simpleType>
</xs:complexType>
</xs:element>

<xs:complexType name="nxt:depPartType">
  <xs:attribute name="xpath">
    <xs:simpleType>
      <xs:annotation>
        <xs:documentation>
          A subset of XPath expressions for use in fields or streamNext {next,current,first}/@xpath
        </xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:token"/>
    </xs:simpleType>
  </xs:attribute>
  <xs:choice>
    <xs:element ref="nxt:value"/>
    <xs:element ref="nxt:comparator"/>
    <xs:element ref="nxt:set"/>
  </xs:choice>
</xs:complexType>

SchemaForSchemas streamNext

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema_
  xmlns:nxt="http://www.dbis.ethz.ch/streamingXQueryP-nxt"
  target-namepace="http://www.dbis.ethz.ch/streamingXQueryP-nxt"
  elementFormDefault="qualified">
  <xs:element name="dependency">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="current" type="nxt:depPartType"/>
        <xs:element name="next" type="nxt:depPartType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
<xs:complexType>
  <xs:element name="comparator">
    <xs:complexType>
      <xs:choice minOccurs="2" maxOccurs="2">
        <xs:element ref="nxt:first"/>
        <xs:element ref="nxt:next"/>
        <xs:element ref="nxt:current"/>
        <xs:element ref="nxt:value"/>
        <xs:element ref="nxt:operator"/>
      </xs:choice>
      <xs:attribute name="type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration>greater</xs:enumeration>
            <xs:enumeration>greaterequal</xs:enumeration>
            <xs:enumeration>equal</xs:enumeration>
            <xs:enumeration>smallerequal</xs:enumeration>
            <xs:enumeration>smaller</xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="operator">
    <xs:complexType>
      <xs:choice minOccurs="2" maxOccurs="2">
        <xs:element ref="nxt:first"/>
        <xs:element ref="nxt:next"/>
        <xs:element ref="nxt:current"/>
        <xs:element ref="nxt:value"/>
        <xs:element ref="nxt:operator"/>
      </xs:choice>
      <xs:attribute name="type">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration>addition</xs:enumeration>
            <xs:enumeration>substraction</xs:enumeration>
            <xs:enumeration>multiplication</xs:enumeration>
            <xs:enumeration>division</xs:enumeration>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="next" type="streamNextxpath">
    <xs:element name="first" type="streamNextxpath">
      <xs:element name="current" type="streamNextxpath">
        <xs:element name="set">
          <xs:complexType>
            ...
<xs:element name="setElement" maxOccurs="unbounded"/>
<xs:complexType>
  <xs:complexContent>
    <xs:attribute name="value" type="xs:string"/>
  </xs:complexContent>
</xs:complexType>
</xs:element>
<xs:complexType>
  <xs:complexContent>
    <xs:attribute name="value" type="xs:string"/>
  </xs:complexContent>
</xs:complexType>
</xs:element>
<xs:complexType name="streamNextxpath">
  <xs:attribute name="xpath">
    <xs:simpleType>
      <xs:annotation>
        <xs:documentation>A subset of XPath expressions for use in fields or streamNext {next, current, first}/@xpath</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:token"/>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
</xs:schema>
A.4 Stream Xerces Sample Program

```
import java.io.IOException;
import javax.xml.XMLConstants;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.validation.SchemaFactory;
import org.apache.xerces.xs.PSVIProvider;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;

public class Validation extends DefaultHandler {
    private static PSVIProvider psviProvider;
    private static XMLReader reader;

    public static void main(String[] args) throws SAXException,
    ParserConfigurationException, IOException {
        Validation handler = new Validation();
        System.setProperty("javax.xml.parsers.SAXParserFactory",
                            "org.apache.xerces.jaxp.SAXParserFactoryImpl");

        SchemaFactory factory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

        // Setup SAX parser for schema validation.
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setNamespaceAware(true);
        spf.setFeature("http://xml.org/sax/features/validation", true);
        spf.setFeature("http://apache.org/xml/features/validation/schema", true);
        spf.setFeature("http://apache.org/xml/features/validation/schema/augment-psvi", true);
        spf.setFeature("http://apache.org/xml/features/validation/warn-on-undeclared-elemdef", true);
        spf.setFeature("http://apache.org/xml/features/streamingMode", true);
        spf.setFeature("http://apache.org/xml/features/validation/schema-full-checking", true);

        SAXParser parser = spf.newSAXParser();
        reader = parser.getXMLReader();
        psviProvider = (PSVIProvider) reader;

        reader.setProperty("http://apache.org/xml/properties/schema/external-schemaLocation",
        reader.setContentHandler(handler);
        reader.parse("todolist.xml");
    }
```
public void characters(char[] ch, int start, int length) throws SAXException {

    /** Start element. */
    public void startElement(String uri, String localName, String qname,
        Attributes attributes) throws SAXException {
        // startElement(String,String,String,Attributes)
    }

    /** End element. */
    public void endElement(String uri, String localName, String qname)
        throws SAXException {
        // endElement(String,String,String)
    }
}
Bibliography