Technical Survey of XML Schema and Query Languages

Angela Bonifati
Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
Email: bonifati@elet.polimi.it

Dongwon Lee
Department of Computer Science, University of California at Los Angeles
Los Angeles, CA 90095, USA
Email: dongwon@cs.ucla.edu

June 22, 2001

Abstract

XML [9] is rapidly becoming one of the most widely adopted technologies for information exchange and representation on the World Wide Web. As XML is emerging as *the* data format of the Internet era, there is a substantial increase of the amount of data encoded in XML format. To better describe such XML data structures and constraints, several XML schema languages have been proposed by academic institutions, Internet-related major companies and the World Wide Web Consortium. Similarly, to better query and update such XML data, novel query languages for extracting, transforming and integrating the XML content have been defined, some in the tradition of database query languages (i.e. SQL, OQL, Datalog), others more closely inspired by XML, and others with a reasonably mixed flavor.

The W3C (http://www.w3c.org) published both a recommendation for a standard schema language (XML-Schema) and a working draft for a standard query language (XQuery). In this paper, we present a detailed state of the art of twelve most representative schema and query languages for XML. We analyze both separate and overlapping key features and differences of schema and query languages respectively, underline the schema and query's different missions, and provide a taxonomy of them, while laying the future expectations and developments.

We expect this work to be useful to both the standardization process and the developers in search of the appropriate language for their applications.

Keywords: XML, Schema Language, Query Language, Language Comparison, Taxonomy

1 Motivations

As more and more information is either stored in XML, exchanged in XML or represented as XML through various interfaces, the ability to structure and query XML data becomes increasingly important. In order to represent such XML data structures and constraints, and to suitably query them, several Schema and Query languages have been proposed by several institutions, Internet-related companies and W3C itself. We have compared and classified the most representative Schema and Query languages: XML-Schema, DSD, Schematron, SOX, RELAX, and XML-DTD as exemplars of Schema languages and Lorel, XQuery, XML-QL, XML-GL, XQL and XSLT as exemplars of Query languages. The result of our analysis is a comprehensive state of the art that depicts the current evolution of these languages from many well-defined perspectives. This work is an extension and integration of our previous work [4, 41], which takes into account new important schema and query language proposals. In particular, for the Schema languages set, we have added the analysis of RELAX, which appeared in 2000, and XML-Schema, which has been endorsed as W3C Recommendation on May, 2001. We aim to highlight which features of

the language are commonly shared, and which ones the standard language lacks or supersedes. For the Query languages set, the extensions of XML-GL, XSLT and XQL have been considered since our previous work. Moreover, in February 2001, the first W3C working draft for a standard query language, XQuery, has been published. The evolution of the standardization process is only at its early stage. Depicting the current situation will help to understand which features of the current proposal are inherited from the past, which are completely new, which are absent or partially covered.

2 Introduction to the Schema and Query Languages

2.1 Six Schema Languages

As of June 2001, about a dozen XML schema languages have been proposed. Among those, in this paper, we choose six schema languages (XML-DTD [9], XML-Schema [3, 58], RELAX [48, 34], SOX [23], Schematron [35, 45], DSD [36, 37]) as representatives with the following reasons: they are backed by substantial organizations so that their chances of survival are high (e.g., XML-DTD and XML-Schema by W3C, RELAX by Japanese Standard Association, DSD by AT&T); there are publicly known usages or applications of them (e.g., XML-DTD in XML, SOX in xCBL[1]); some languages have a unique approach distinct from XML-DTD (e.g., SOX, Schematron, DSD). In the following, we briefly review each schema language.

2.1.1 XML-DTD

XML-DTD (DTD in short), a subset of SGML DTD, is the *de facto* standard XML schema language of the past and present and is most likely to thrive until XML-Schema finally arrives. It has limited capabilities, compared to other schema languages. Its main building block consists of an *element* and an *attribute*. The real world is typically represented by the use of hierarchical element structures. The 2nd Edition of the language specification became a W3C Recommendation on October 6, 2000.

2.1.2 XML-Schema

According to [43], XML-Schema language is intended to be more expressive than DTD while it can be expressed in XML notations and more usable by a wider variety of applications. It has many novel mechanisms such as inheritance for attributes and elements, user-defined datatypes, etc. XML-Schema became W3C Recommendation on May 2, 2001. The specification is stable and has been reviewed by the W3C Members, who favors its adoption by academic, industry, and research communities.

2.1.3 SOX

SOX (Schema for Object-Oriented XML) is an alternative schema language for defining the syntactic structure and partial semantics of XML document types. SOX leverages on the object-oriented relationships between data structures to allow the easy management of a component library, and the use of type relationships within schema-based e-commerce applications. The current version, 2.0, is developed by Commerce One. The xCBL (XML Common Business Library) 3.0 is implemented using SOX version 2.0.

2.1.4 Schematron

Schematron, created by Rick Jelliffe, is quite unique among the others, since it focuses on *validating* schemas using patterns instead of *defining* schemas. Its schema definition is simple enough to be defined in a single page, and it still provides powerful constraint specification via XPath [16]. The latest version is 1.5.

[1] http://www.xcbl.org/

2.1.5 DSD

DSD 1.0 was co-developed by AT&T Labs and BRICS with the goals of context-dependent description of elements and attributes, flexible default insertion mechanisms, expressive power close to XSLT [18], etc. Like Schematron, DSD puts a strong emphasis on *schema constraints*. DSD has been used in an AT&T application, named XPML. A revised specification, DSD 1.1, is under development.

2.1.6 RELAX

RELAX (REgular LAnguage description for XML) is a novel XML schema language developed by Makoto Murata. RELAX has been standardized by JSA (Japanese Standard Association) and was submitted to ISO (International Standard Organization) as a fast track procedure document for DTR processing and recently has been approved by the ISO/CS ballot [34]. RELAX is based on clean principles of hedge automata [47] and has the capability of expressing context-sensitive schema rules by means of non-terminal symbols. At the time of writing, RELAX and TREX [15] are being merged into RELAX NG [17] under OASIS Technical Committee.

2.2 Six Query Languages

As of June 2001, several XML query languages have been proposed. Among those, in this paper, we choose six query languages (Lorel [2], XML-QL [26], XML-GL [12], XSLT [18], XQL [54], XQuery [13]) as representatives. Lorel has been chosen because it can be considered a suitable representative of the family of languages for semistructured data. XML-QL is the first query language in XML syntax and has forth contributed to the development of Quilt and of XQuery. XML-GL is a visual language addressing the complexity of XML data. XSLT is a very popular transformation language, which is properly backed by efficient and full-fledged processors. XQL inherits most of XPath [16] patterns and is very simple and easy to learn. XQuery is the first proposal of a W3C standard query language for XML, which embeds the experience of previously defined query languages. In the following, we briefly review each query language, showing their creation date, their current development (if any) and their purpose.

2.2.1 Lorel

Lorel[2] was originally designed for querying semistructured data [1] and has now been extended to XML data [1]; it was conceived and implemented at Stanford University. It is a user-friendly language in the SQL/OQL style, it includes a strong mechanism for type coercion and permits powerful path expressions, useful when the structure of a document is not known in advance [2].

2.2.2 XML-QL

XML-QL[3] was designed at AT&T Labs; it has been developed as part of Strudel Project. XML-QL language extends SQL with an explicit `CONSTRUCT` clause for building the document resulting from the query and uses the *element patterns* (patterns built on top of XML syntax) to match data in an XML document. XML-QL can express queries as well as transformations, for integrating XML data from different sources [26, 27].

2.2.3 XML-GL

XML-GL is a graphical query language, relying on a graphical representation of XML documents and DTDs by means of labeled *XML graphs*. It was designed at Politecnico di Milano; an implementation is ongoing and a web site is under construction [4]. All the elements of XML-GL are displayed visually;

[2] http://www-db.stanford.edu/lore
[3] http://www.research.att.com/sw/tools/xmlql
[4] http://xerox.polimi.it/Xml-gl

therefore, XML-GL is suitable for supporting a user-friendly interface (similar to QBE) [12]. The last specification of the language can be found in [21].

2.2.4 XSLT

The Extensible Stylesheet Language (XSL) has facilities that could serve as a basis for an XML query language, XSLT (Extensible Stylesheet Language Transformations). An XSLT stylesheet consists of a collection of template rules; each template rule has two parts: a pattern which is matched against nodes in the source tree and a template which is instantiated to form part of the result tree. XSLT makes use of the expression language defined by XPath [16] for selecting elements for processing, for conditional processing and for generating text. It was designed by the W3C XSLT working group [18, 56, 31]. XSLT is still under development at the time of writing. XSLT 1.0 [18] is a W3C recommendation, while the new version is a (XSLT 1.1) working draft [19]. Future developments are described in the requirements specification of XSLT 2.0 [46]. We refer to the version 1.1 throughout the paper and explicitly indicate the number of version, when different.

2.2.5 XQL

XQL is a notation for selecting and extracting XML data. XQL can be considered a natural extension to the XSLT pattern syntax; it is designed with the goal of being syntactically simple and compact (a query could be part of a URL), with a reduced expressive power [53, 51, 52, 50, 54, 56]. The language has been reviewed in 1999 [54].

2.2.6 XQuery

XQuery is the first W3C proposal for a standard query language, published in February 2001 and revised in June 2001 [13]. The current proposal of XQuery version 1.0 is mostly drawn from Quilt [14], a newly-conceived query language for XML, which inherits the experience of several past query languages and attempts to unify them. XQuery assembles many features from previously defined languages, such as the syntax for navigation in hierarchical documents from XPath and XQL, the notion of binding variables from XML-QL, the combination of clauses a-la SQL and the notion of a functional language from OQL; it is designed with the goal of being expressive, of exploiting the full versatility of XML and of combining information from diverse data sources [13].

2.3 Other Schema and Query Languages

Other schema languages include EXPRESS [33], DCD [6], XDR [30, 40, 44], DDML [5], Assertion Grammars [49], DT4DTD [11], TREX [15], and RELAX NG [17]. Other query languages include XMAS [42], Quilt [14], YATL [20] and XDuce [32].

2.4 Outline

The paper is organized as follows: in Section 3, we describe how the languages support various orthogonal features; in Sections 3.1 and 3.2, from various perspectives, we compare the features of those schema and query languages, respectively. Section 4 summarizes the desired qualities of those languages, and proposes a relative ranking of them. Concluding Section 5 proposes a language taxonomy which shows, in a synthetic and effective way, a comparison of the expressive power of the dozen XML schema and query languages.

3 Features Classification

In comparing the languages, we have identified categories of features, separately for schema languages and query languages. The features of the six schema languages are organized into 7 categories: inte-

gration with XML, typing & extensibility, basic schema abstractions, order management, constraints, schema evolution, and miscellaneous. The features of the six query language are organized into 10 categories: data model, basic query abstractions, path expressions, quantification, negation, reduction & filtering, restructuring abstractions, aggregation, nesting and set operations, order management, typing & extensibility, integration with XML, and update language.

The overlap among the categories of the schema and query language is minimal, due to their quite different missions. A schema language is devoted to define the structure, content and semantics of XML documents and to enforce constraints to XML data. Conversely, a query language is expected to accomplish data extraction and transformation. We have highlighted the common abstractions of schema and query in the `Basic Schema Abstractions` and `Basic Query Abstractions` categories. Moreover, `Integration with XML`, `typing & extensibility` and `order management` are commonly shared between the set of Schema and Query languages.

When a schema or query language supports a certain feature fully or partially, we denote it as `Yes` or `Partial`. More precisely, a feature is partially supported when it is not completely covered by the language (i.e., because the language can specify a portion of the whole feature). Otherwise, we denote it as `No`. Conceivably, when there is no explicitly equivalent construct in the language, but the feature can be simulated using other constructs with reasonable complexity, we consider the feature as supported by the language. For the sake of clarity, we denote a constant value within single quotes even if it is not supported by the language specification. Furthermore, any attribute A or element E in the language is denoted by ⟨A⟩ or ⟨E⟩.

3.1 Six Schema Languages

3.1.1 Integration with XML

In this section, the syntax of schema languages and their support for namespace are discussed.

1. *Syntax in XML.* Using XML syntax for the schema language brings several benefits [39]: first, users do not have to learn new proprietary syntax; then, the schema language can be readily applicable to existing XML applications (e.g., editors, browsers); finally, the schema file can be stored in an XML storage system along with XML documents, and the schema language can be easily extended. All the schema languages except DTD are written in XML syntax:

DTD: `No` XML-Schema: `Yes` RELAX: `Yes` SOX: `Yes` Schematron: `Yes` DSD: `Yes`

2. *Namespace.* XML namespace provides a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references [7] Some languages support the declaration of the namespaces as follows:

DTD: `No` XML-Schema: `Yes` RELAX: `No` SOX: `Yes` Schematron: `Yes` DSD: `No`

DTD does not support namespaces. Both DSD and RELAX plan to support namespaces, but the specification is not available at the time of writing. All other languages support namespaces. Suppose one wants to define the `book` element by reusing the `address` element (defined elsewhere and denoted as `URI`) and defining his own `title` element. This can be written in XML-Schema as follows:

```
<schema xmlns:z='URI' ...>
  <element name='book'>
    <complexType>
      <element name='title' type='string'/>
      <element name='address' type='z:address'/>
    </complexType>
  </element>
</schema>
```

SOX supports the element ⟨Namespace⟩ to declare namespace and two attributes ⟨Prefix⟩ and ⟨Type⟩ to qualify names.

```
<namespace prefix='z' namespace='URI'/>
<elementtype name='book'>
  <model>
    <element type='title'/>
    <element prefix='z' type='address'/>
  </model>
</elementtype>
```

Similarly, Schematron provides an attribute Ns for the element ⟨schema⟩.

3. *Include & import.* Sometimes, it is convenient to embed externally defined schema fragments in the current schema. When the schema gets larger, it becomes more desirable to have modular schema definitions for guaranteeing better maintainence and readability. Several schema languages support this feature. If the newly embedded fragments can have only the *same* target namespace as the current schema, then they will be referred through the directive *include*. In all the other cases, they will be referred through the directive *import*. The directive *include* is supported as follows:

DTD: No XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: No DSD: Yes

In XML-Schema, using ⟨Include schemaLocation='URI'⟩ is conceptually equivalent to replacing the include clause with all the definitions in the URI. The namespace of the included fragments must be the same as that of the current schema. RELAX provides an element ⟨Include moduleLocation='URI'⟩ for the same purpose. In SOX, a construct ⟨Join⟩ allows schema definitions belonging to the same namespace to be embedded. Similarly, ⟨Include⟩ is supported in DSD as well. The directive *Import* is supported as follows:

DTD: No XML-Schema: Yes RELAX: No SOX: Yes Schematron: No DSD: No

In XML-Schema, a construct ⟨Import⟩ exists. By importing multiple namespaces, XML-Schema allows definitions and declarations contained in schemas under different namespaces. In SOX, special processing instruction ⟨? import ?⟩ is used to import schema that can override the default namespace declared in the current schema.

3.1.2 Typing & Extensibility

In this section, we discuss about typing and extensibility features of the schema languages. Type-checking is an important feature of a schema language and can dramatically reduce the burden of application programs. Conceivably, extensibility allows type derivation from existing types.

Basically, datatypes are computer representations of well known abstract concepts such as numbers and dates. More specifically, datatypes are 3-tuple, consisting of 1) a set of distinct values, 2) a set of lexical representations, and 3) a set of facets that characterize properties of the values or lexical items [3]. There are multiple inconsistent definitions of what the datatypes are among different languages. In this paper, we chose a notation, which exhibits any properties in 3-tuple. Datatype can be in general categorized into two types: *simple* or *complex*. A *simple* type cannot have element content nor carry attributes while a *complex* type can have both. Although most schema languages support simple types separately, the support of complex types is a bit fuzzy due to the mixed definition of complex types and element types.

Types can be derived by inheritance. As in the object-oriented paradigm, inheritance is done by *extending* or *restricting* the base types. In this section, we have divided the target of inheritance with respect to *simple* and *complex* types. When some languages support inheritance toward *attribute* and *element* instead, we classify it as simple and complex type inheritance, respectively.

1. *Built-in type.* This is either a primitive or a derived *simple* type provided by the schema language specification. Most schema languages, except Schematron and DSD, support an array of built-in types including the plain string and XML-related types (e.g., ID, NMTOKEN). The numbers of such built-in types

are:

DTD: 10 XML-Schema: 45 RELAX: 45 SOX: 17 Schematron: 0 DSD: 6

While DTD supports only XML-related primitive types (i.e., CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION, PCDATA), XML-Schema, SOX and RELAX support an extensive set of built-in types, covering the most common types used in general-purpose programming languages. RELAX completely borrows its datatypes from XML-Schema. Since the focus of Schematron is on validating XML structure, it does not provide any explicit built-in type. Similarly, DSD provides types Char, YesOrNo, Numeral, StringType , ID, and IDRef. However, many built-in types can be easily simulated by means of regular expressions in DSD.

2. *User-defined type.* A capability to specify user-defined types greatly enhances the flexibility of the schema language.

DTD: No XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Partial DSD: Yes

XML-Schema, SOX, and DSD provide the facility of defining *simple* types in the schemas. Since RELAX borrows its typing feature from XML-Schema, it also has the capability. In XML-Schema, new simple types can be created by deriving from built-in or derived types using the inheritance primitives. Details will be found in Section 3.1.2. In SOX, new datatypes can be defined using three *facets* (⟨Enumeration⟩, ⟨scalar⟩ and ⟨varchar⟩). Although types can be simulated in Schematron, they are not considered as first-class objects as in the other languages. DSD uses a construct ⟨StringTypeDef⟩ along with a rich set of operators and regular expressions to support user-defined types. For instance, in DSD a 9 digit US `zipcode` definition can be written as follows:

```
<StringTypeDef ID='zipcode'>
  <Sequence>
    <Repeat value='5'/>
      <CharSet Value='0123456789'/>
    </Repeat>
    <Optional>
      <String Value='-'/>
      <Repeat value='4'/>
        <CharSet Value='0123456789'/>
      </Repeat>
    </Optional>
  </Sequence>
</StringTypeDef>
```

3. *Type domain constraint.* The domain constraint of a type allows to define the legal values admitted for that type. Some languages support a set of constructs to limit the valid domain values for datatypes.

DTD: No XML-Schema: Yes RELAX: Yes SOX: Partial Schematron: Yes DSD: Yes

With respect to this feature, XML-Schema supports a multitude of facets (e.g., range, precision, length, mask) and regular expressions. RELAX again borrows all facets from XML-Schema. SOX provides a primitive set of facets including enumeration, min or max value, maxlength, etc. However, SOX does not support a pattern language, which could have augmented the expressiveness of the language substantially. Although built-in or user-defined types are not allowed in Schematron, it is possible to simulate such types using Schematron's support of XPath. For instance, in Schematron the integer type for the element `E` can be simulated as follows [45]:

```
<rule context='E'>
  <assert test='floor(.) = number(.)'>
    E can have only integer value.</assert>
</rule>
```

As already shown in the example of the user-defined type, DSD supports a set of pattern-related operators to constrain the legal domain for user-defined types.

4. *Explicit null.* A desired feature on datatypes is to differentiate among unknown, absent and available data by supporting the explicit "null" values.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: No DSD: No

In XML-Schema, there is an attribute ⟨Nullable⟩ to indicate that the element content is null. In an XML instance document, the element **fullname** carries an attribute **null='true'** to represent the null value as shown below:

```
XML-schema  : <element name='fullname' nullable='true'/>
instance: <fullname xsi:null='true'></fullname>
```

5. *Simple type by extension.* New simple types may be created by deriving from other simple types with more relaxed domain constraints. The set of legal values of the new type is a *superset* of that of the base type. None of the languages supports this feature.

DTD: No XML-Schema: No RELAX: No SOX: No Schematron: No DSD: No

6. *Simple type by restriction.* The set of legal values of the new type is a *subset* of that of the base type.

DTD: No XML-Schema: Yes RELAX: No SOX: Yes Schematron: No DSD: No

In XML-Schema, inheritance among simple types is allowed as shown in the following example, where a 9 digit US **zipcode** is created from the base type **string**:

```
<simpleType name='zipcode' base='string'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

By constraining the domain values by means of the pattern expression, the legal values for the **zipcode** have been restricted from the **string** type. In SOX, new datatypes may be *refined* from built-in or derived types. For instance, the new datatype **RGB** allows only three values from the **color** type.

```
<datatype name='RGB'>
  <enumeration datatype='color'>
    <option>Red</option> <option>Green</option> <option>Blue</option>
  </enumeration>
</datatype>
```

7. *Complex type by extension.* This feature derives a new type by extending an old type.

DTD: No XML-Schema: Yes RELAX: No SOX: Yes Schematron: No DSD: No

XML-Schema supports type inheritance using the construct ⟨Extension base='SomeBaseType'/⟩. Newly added elements are always appended at the end. Typically, one indicates that the content models of the new types are complex, i.e., they contain elements, declared by means of the ⟨ComplexContent⟩ element. In SOX, ⟨Extends type='basetype'⟩ is supported, where *appending* new elements and attributes is also feasible. Given the **person** element defined elsewhere, the following example illustrates how the new element **new-person** inherits the content model of the **person** element and has an additional element **address** and an attribute **email**.

```
<elementtype name='new-person'>
  <extends type='person'>
    <append>
      <element name='address' type='addr'/>
```

```
    </append>
    <attdef name='email' datatype='string'/>
  </extends>
</elementtype>
```

In DSD, any definition can be *redefined* using the ⟨RenewID⟩ and ⟨CurrIDRef⟩ constructs. However, once the new type is defined, the original type is no longer accessible. Therefore, this capability serves the need of *renewing* rather than of *deriving*.

8. *Complex type by restriction.* This feature derives a new type by restricting an old type.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: No DSD: No

In XML-Schema, it is possible to derive new types by restricting the content models of existing types. The values admitted by the new type are a subset of the values admitted by the base type. For instance, the following schema shows how the newly defined element type `NewItemType` is required to have at least 100 `item` sub-elements as a new restriction.

```
<complexType name='ItemType'>
  <sequence>
    <element name='item' minOccurs='0'/>
    <element name='price'/>
  </sequence>
</complexType>

<complexType name='NewItemType'>
  <restriction base='ItemType'>
    <sequence>
      <element name='item' minOccurs='100'/>
      <element name='price'/>
    </sequence>
  </restriction>
</complexType>
```

Notice that types derived by restriction must repeat all the components of the base type definition.

3.1.3 Basic Schema Abstractions

In this section, we discuss about the features constraining the structure of schema.

1. *Attribute default value.* When a value is not present in an XML document, a pre-determined default value is inserted.

DTD: Yes XML-Schema: Yes RELAX: No SOX: Yes Schematron: No DSD: Yes

RELAX and Schematron do not support default values. In RELAX, this feature is intentionally omitted in order to maintain the compatibility with the existing XML processors. Consequently, default values are left to the responsibility of application programs. All others support default values as follows: In an attribute declaration of DTD, if the declaration is neither `#REQUIRED` nor `#IMPLIED`, then the attribute value contains the declared default value.

```
<!ATTLIST list  type   (bullets|ordered) 'ordered'>
<!ATTLIST form  method CDATA #FIXED 'POST'>
```

Here, the attribute `type` of the element `list` has a default value of "ordered" while the attribute `method` of the element `form` has a fixed value of "POST". Other languages support default values similarly. The following two snippets of XML-Schema and SOX respectively illustrate an attribute `nm` with a default value "John Doe":

```
<attribute name='nm' use='default' value='John Doe'/>
```

```
<attrdef name='nm' datatype='string'>
  <default>John Doe</default>
</attrdef>
```

DSD provides a more sophisticated way of defining defaults for attributes by associating them with a boolean expression. For instance, in DSD, one can specify a default value of "John Doe" for male employees as follows:

```
<ElementDef ID='employee'>
  <AttributeDecl Name='nm'/>
  <AttributeDecl Name='gender'/>
</ElementDef>
...
<Default>
  <Context><Element Name='employee'>
    <Attribute Name='gender' Value='M'/>
  </Element></Context>
  <DefaultAttribute Name='nm' Value='John Doe'/>
</Default>
```

2. *Choice among attributes.* This feature describes a constraint on the presence of attributes that are selected from a given list.

DTD: No XML-Schema: No RELAX: No SOX: No Schematron: Yes DSD: Yes

Schematron and DSD can express the requirement that exactly one of the two attributes `fn` and `gn` must be present as an attribute of `person` element, as it appears from these examples:

```
<rule context='person'>
  <assert test='@fn or @gn'>Or semantics</assert>
  <assert test='count(attribute::*) = 1'>
    Only one attribute</assert>
</rule>

<ElementDef ID='person'>
  <AttributeDecl Name='fn' IDType='ID'/>
  <AttributeDecl Name='gn' IDType='ID'/>
  <OneOf>
    <Attribute Name='fn'/><Attribute Name='gn'/>
  </OneOf>
</ElementDef>
```

3. *Optional vs. required attributes.* In all languages, it is possible to express whether or not an attribute definition in an XML document instance is either required or optional.

DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Yes DSD: Yes

To denote that an attribute must be present, DTD uses a keyword `#REQUIRED` while XML-Schema uses ⟨Use='required'⟩ in the attribute declaration. Similarly, in RELAX, an attribute ⟨Required='true'⟩ is used while in SOX, an element ⟨Required/⟩ is used for mandatory attribute definition. Schematron can enforce this feature using a pattern ⟨Assert test='@attribute-name'⟩. Like RELAX, DSD supports an attribute ⟨Optional='no'⟩.

4. *Attribute domain constraint.* Some languages can specify legal values for attributes.

DTD: Partial XML-Schema: Yes RELAX: Yes SOX: Partial Schematron: Yes DSD: Yes

DTD and SOX provide only the enumeration capability by which users can list all legal values for the attribute being defined. For instance, the following snippets show examples of DTD and SOX for an enumerated attribute type, `RGB`:

```
<!ATTLIST spec RGB (red|green|blue)>

<elementtype name='spec'>
  <attrdef name='RGB' datatype='string'>
    <enumeration datatype="NMTOKEN">
      <option>red</option>
      <option>green</option>
      <option>blue</option>
    </enumeration>
  </attrdef>
</elementtype>
```

In XML-Schema, domain values for simple types can first be constrained using various facets and then new attributes can be defined using simple types. In RELAX, constraints such as ⟨Enumeration⟩, ⟨Mininclusive⟩, ⟨Maxinclusive⟩ can be applied to attributes. In Schematron, the support for an arbitrary domain constraint rule for attribute values is possible as shown in the case of the domain constraint for datatypes in Section 3.1.2. In DSD, one can apply numerous operators such as ⟨Union⟩ or ⟨Repeat⟩ to the construct ⟨StringType⟩ to constrain domain values.

5. *Element default value.* Elements can have either *simple* or *complex* default values.

DTD: No XML-Schema: Partial RELAX: No SOX: No Schematron: No DSD: Yes

In XML-Schema, one can provide a string value as the default value when the element has a simple type, but cannot provide defaults values for complex types.

```
<element name='fullname' type='string' default='John Doe'/>
```

DSD allows both simple and complex default values for elements using ⟨DefaultContent⟩. For instance, one can specify that a default `address` is "Los Angeles" and "CA":

```
<Default>
  <Context><Element Name='address'/></Context>
  <DefaultContent>
    <city>Los Angeles</city><state>CA</state>
  </DefaultContent>
</Default>
```

6. *Element content model.* The element content model can be empty, text (including datatype), element, or mixed (text plus element).

DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Partial Schematron: Yes DSD: Yes

DTD supports all four content models as follows:

```
empty   : <!ELEMENT o EMPTY>
text    : <!ELEMENT p (#PCDATA) >
element : <!ELEMENT q (x?|y*|z+) >
mixed   : <!ELEMENT r (#PCDATA|x)* >
```

Similarly, XML-Schema supports the four content models as shown below. Note that the "empty" content model in XML-Schema is implicitly specified by providing no element type and child elements.

```
empty   : <element name='o'/>
text    : <element name='p' type='string'/>
element : <element name='q'>
            <complexType>
              <choice>
                <element name='x' minOccurs='0' maxOccurs='1'/>
                <element name='y' minOccurs='0' maxOccurs='unbounded'/>
                <element name='z' minOccurs='1' maxOccurs='unbounded'/>
```

```
              </choice>
            </complexType>
          </element>
mixed   : <complexType name='rType' type='mixed'>
            <choice minOccurs='0' maxOccurs='unbounded'>
              <element name='x'/>
            </choice>
          </complexType>
          <element name='r' type='rType'/>
```

RELAX also supports all four models as follows[5]:

```
empty   : <elementRule role='o' type='emptyString'/>
text    : <elementRule role='p' type='string'/>
element : <elementRule role='q'>
            <choice>
              <element name='x' occurs='?'/>
              <element name='y' occurs='*'/>
              <element name='z' occurs='+'/>
            </choice>
          </elementRule>
mixed   : <elementRule role='r'>
            <mixed>
              <choice occurs='*'>
                <element name='x'/>
              </choice>
            </mixed>
          </elementRule>
```

SOX only supports three content models using constructs ⟨Empty/⟩, ⟨string/⟩ and ⟨Element/⟩ respectively, but does not explicitly support the mixed content model. In Schematron, the following XPath expression can be used as a value for ⟨Assert⟩ construct to specify the four content models:

```
empty   : not(*)
text    : string-length(text()) > 0
element : count(element::*) = count(*)
mixed   : by default
```

DSD also supports all the four models using the constructs ⟨Empty/⟩, ⟨StringType/⟩, ⟨Element/⟩ and ⟨AnyElement/⟩.

7. *Choice among elements.* Only one sub-element among candidates is allowed.

   DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Yes DSD: Yes

DTD uses an operator "|" to denote choice among elements. Using a grouping construct ⟨choice⟩ in XML-Schema, one can specify that only one of the sub-elements in the group must appear. Both RELAX and SOX support the ⟨Choice⟩ content model for an element. Schematron can express its choice among elements using rules similar to the case of choice among attributes, as shown in Section 3.1.3. In DSD, the construct ⟨OneOf⟩ is supported as follows:

```
<ElementDef ID='person'>
  <OneOf>
    <Element Name='fn'/><Element Name='gn'/>
  </OneOf>
</ElementDef>
```

[5]For simplicity, we used a RELAX element **element** instead of **ref**, **elementRule**, and **tag**. The keyword **element** must be considered syntactic sugar.

8. *Min & Max occurrence.* This feature describes whether the languages can set up the minimum occurrence and maximum occurrence of element (and content model) instances.

DTD: Partial XML-Schema: Yes RELAX: Partial SOX: Yes Schematron: Yes DSD: Partial

In DTD, the occurrences of elements can be only primitively controlled by the three operators: "?" for 0 or 1, "$*$" for 0 or many and "+" for 1 or many. RELAX uses the same approach; it allows the three operators as values of ⟨Occurs⟩ attribute. In XML-Schema, an element declaration carries attributes such as MinOccurs='k' and MaxOccurs='l'. In SOX, an element definition carries ⟨Occurs⟩ attribute that indicates the number of repetitions of the instantiated element. It can take the three Kleene operators (i.e., ?, $*$, +), and use it in place of the maximum cardinality ("k,l", or "k,$*$"). In Schematron, this can be written as ⟨Assert test='count(E)>=k'⟩ and ⟨Assert test='count(E)<=l'⟩. In DSD, the occurrences of elements can be specified as ⟨Optional⟩, ⟨ZeroOrMore⟩, ⟨OneOrMore⟩, and ⟨Union⟩, but expressions with any value of the minimum and maximum numbers cannot be expressed.

3.1.4 Order Management

1. *Ordered list.* This feature investigates whether the order among sub-elements must be preserved to ensure document validness.

DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Yes DSD: Yes

In DTD, sub-elements listed with an operator "," must respect the order in which they appear. Likewise, in XML-Schema, the order needs to be preserved unless otherwise specified. One can also explicitly specify a sequential order by using a grouping construct ⟨Sequence⟩. Both RELAX and SOX support ⟨Sequence⟩ content models as well. For instance, the following examples show RELAX and SOX at work on the order content of the `person` element, which is enforced to contain the sub-element `fn` followed by the sub-element `ln`:

```
<elementRule role='person'>
  <sequence>
    <element name='fn' type='string'/>
    <element name='ln' type='string'/>
  </sequence>
</elementRule>
<tag name='person'/>

<elementtype name='person'>
  <model><sequence>
    <element name='fn'/><element name='ln'/>
  </sequence></model>
</elementtype>
```

The same schema can be written in Schematron as follows:

```
<rule context='person'>
  <assert test='(*[position()=1] = fn) and (*[position()=2] = ln)'>
    fn must be followed by ln.</assert>
</rule>
```

The ordered list in DSD is expressed in a similar fashion by the construct ⟨Sequence⟩ in an element content definition.

```
<ElementDef ID='person'>
  <Sequence>
    <Element Name='fn'/><Element Name='ln'/>
  </Sequence>
</ElementDef>
```

2. *Unordered list.* This feature expresses the support of declared unorderness among sub-elements. This is connatural in some languages, but more difficult to grasp in others.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: Yes DSD: Yes

As opposed to SGML, which offers an operator "&" to create an unordered list, DTD does not offer an explicit operator for unordered lists. Instead, one needs to encode all the possible combinations of the sub-elements. For instance, to express an unordered list of sub-element (a & b & c) of SGML in DTD, one has to write ((a,b,c)|(a,c,b)|(b,a,c)|(b,c,a)|(c,a,b)|(c,b,a)) or an incorrect expression like (a|b|c)* [55]. By using a grouping construct ⟨All⟩, one can specify the unordered list in XML-Schema.

In Schematron, if no pattern has been specified, the unordered list is obtained by default. In DSD, a single content expression describes a set of allowed lists of string data and elements. Several content expressions describe all *merging* of lists, one from each expression. Thus, by cleverly using this feature, one can capture "floating elements", i.e., mixes of ordered and unordered contents. This feature in DSD is more expressive than the simple ordered or unordered content model.

3.1.5 Constraints

Constraints play an important role in enforcing the correct behavior of applications. By allowing schema languages to specify detailed constraints for a schema, application programs do not have to check the correctness of the semantics of the schema. We first, consider how schema languages allow to describe uniqueness or keyness. The discussed uniqueness or keyness features cover two dimensions: scoping (global or local) and compoundness (attribute, element or both). Second, as a more sophisticated constraint specification, we analyze situations in which elements or attributes are allowed only in certain situations (or *contexts*). There are mainly three types of context-sensitive rules found in XML schema languages: attribute definitions depending on the value of other attributes; element (or content model) definitions depending on the value of other attributes, and, finally, element (or content model) definitions depending on the value of parent elements. We will denote each as "Attr-Attr", "Elm-Attr", and "Elm-Elm", respectively.

1. *Uniqueness for attribute.* In general, if there must be only one instance of a type, then the type needs to be defined as *unique*.

DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Yes DSD: Yes

All the languages fully support this feature. To declare unique attributes, DTD, RELAX, SOX and DSD make use of the `ID` type, while XML-Schema uses the keyword ⟨Unique⟩, while the constructs ⟨Selector⟩ and ⟨Field⟩ respectively allow to specify the scope and target object of the uniqueness. Since Schematron does not have an explicit construct such as `ID` of DTD, uniqueness for an attribute must be simulated using pattern "`count()=1`".

2. *Uniqueness for non-attribute.* Schema languages like XML-Schema, Schematron, or DSD allows to specify uniqueness not only for attributes but also for arbitrary elements or composite objects (attribute plus element).

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: Yes DSD: No

This feature can be easily expressed in XML-Schema using the same construct ⟨Unique⟩. For instance, the following schema ensures that there exists a unique `phone` element under `addr` sub-elements of the `person` element.

```
<unique> <selector>person/addr</selector> <field>phone</field> </unique>
```

In Schematron, the same constraint can be written as follows:

```
<rule context='person/addr'>
```

```
  <assert test='count(phone) = 1'>phone is not unique.</assert>
</rule>
```

3. *Key for attribute.* In databases, being a key requires being both unique and not null. XML-Schema supports this feature natively, while other languages support this feature by simulation.

DTD: Yes XML-Schema: Yes RELAX: Yes SOX: Yes Schematron: Yes DSD: Yes

In DTD, by defining an attribute being both `ID` and `#REQUIRED`, one can simulate the same effects of key in databases; Using almost identical syntax as ⟨Unique⟩, a construct ⟨Key⟩ can specify an attribute as a key in XML-Schema. Since RELAX, SOX, and DSD support an attribute with `ID` type, they can also simulate the effects of key as DTD does. In Schematron, this feature can be simulated as follows:

```
<rule context='person'>
  <assert test='@ssn and count(@ssn) = 1'>Is ssn unique?</assert>
  <assert test='string-length(@ssn) > 0'>Is ssn not empty?</assert>
</rule>
```

4. *Key for non-attribute.* "Keyness" is extended for non attributes.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: Yes DSD: No

XML-Schema allows specification of arbitrary elements or composite objects as key. For instance, the following schema in XML-Schema defines the combination of an employee's department code (element) and employee's name (attribute) as a key.

```
<key name='ekey'>
  <selector xpath='employee'/>
  <field xpath='dept/code'/><field xpath='@name'/>
</key>
```

Similarly, Schematron supports this feature by using patterns.

5. *Foreign key for attribute.* *Foreign key* describes both the attributes referencing keys and the attributes referenced by the key.

DTD: Partial XML-Schema: Yes RELAX: Partial SOX: Partial Schematron: Yes DSD: Yes

Similarly to the `ID` type, DTD, RELAX, SOX and DSD use the `IDREF` type for an attribute referencing a key. XML-Schema uses ⟨Keyref⟩. In addition, XML-Schema and DSD support a method (⟨Refer⟩ and ⟨PointsTo⟩, respectively) to specify whom the foreign key actually points to. Furthermore, DSD allows associations of arbitrary boolean expressions with the ⟨PointsTo⟩ construct. For instance, in DSD one can specify "an attribute A that points to either attribute B in an element E_1 or to attribute C in element E_2". In Schematron, this feature can be expressed by means of patterns. The following schema states that `dno` attribute of `employee` element should reference the unique identifier of `dept` element.

```
<rule context = 'employee[@dno]'>
  <assert test='(name(id(@dno)) = 'dept')'>Error occurred.</assert>
</rule>
```

6. *Foreign key for non-attribute.* "Foreign keyness" is extended for non attributes.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: No DSD: Yes

Similarly to specifying uniqueness for non-attributes, XML-Schema can specify foreign keys for arbitrary elements or composite objects using the same ⟨Keyref⟩ construct.

```
<keyref refer='ekey'>
  <selector xpath='project'/>
  <field xpath='emp-dept'/><field xpath='@ename'/>
</keyref>
```

The following DSD example illustrates that an attribute `book-ref` is referencing an element `book`.

```
<AttributeDecl ID='book-ref' IDType='IDRef'>
  <PointsTo>
    <Context><Element Name='book'/></Context>
  </PointsTo>
</AttributeDecl>
```

7. *Attr-Attr.* Often, an attribute a_1 of an element E is relevant only when an attribute a_2 of an element E is defined or has a certain value.

DTD: No XML-Schema: No RELAX: No SOX: No Schematron: Yes DSD: Yes

For instance, the following Schematron schema states that if the element `E` has the attribute `one`, then it must have the second attribute `two` as well:

```
<rule context='E'>
  <report test='(@one) or not(@one and @two)'>
    E cannot have attribute 'one' alone.</report>
</rule>
```

DSD easily supports this feature by using its rich set of boolean operators. For instance, the following snippet states that the `salary` attribute is defined only when the `student` is a "TA":

```
<ElementDef ID='student'>
  <If><Attribute Name='TA' Value='yes'>
    <Then><Optional>
      <AttributeDecl Name='salary'/>
    </Optional></Then>
  </If>
</ElementDef>
```

8. *Elm-Attr.* In this kind of context-sensitivity rule, the element content is controlled by the value of an attribute. For instance, suppose an element `val` has an attribute `type` whose value can be either "integer" or "string". Then, depending on the value of `type` attribute, the content model of the element `val` must be defined accordingly. The correctness of the following XML code must be validated by the processor:

```
legal:   <val type='integer'>1000</val>
legal:   <val type='string'>thousand</val>
illegal: <val type='integer'>thousand</val>
```

This feature is supported as follows:

DTD: No XML-Schema: No RELAX: Yes SOX: No Schematron: Yes DSD: Yes

RELAX supports it by providing two ⟨ElementRule⟩ models with the same Label, but different Roles, as shown in the following:

```
<tag name='val' role='val-integer'>
  <attribute name='type' type='NMTOKEN' required='true'>
    <enumeration value='integer'/>
  </attribute>
</tag>
<elementRule role='val-integer' label='val' type='integer'/>

<tag name='val' role='val-string'>
  <attribute name='type' type='NMTOKEN' required='true'>
    <enumeration value='string'/>
  </attribute>
</tag>
<elementRule role='val-string' label='val' type='string'/>
```

DSD supports this feature using its boolean operators. For instance, the following snippet states that `salary` element is defined only when the `student` element has a `TA` attribute:

```
<ElementDef ID='student'>
  <If><Attribute Name='TA'>
    <Then><Optional>
      <Element IDRef='salary'>
    </Optional></Then>
  </If>
</ElementDef>
```

9. *Elm-Elm.* In this case, element E_1's content is controlled by the parent element E_2 in which E_1 is nested. This feature is supported as follows:

DTD: No XML-Schema: No RELAX: Yes SOX: No Schematron: Yes DSD: Yes

For instance, the following schema in RELAX notation describes two context-sensitivity rules: one, that states that a `para` element contained in a `section` element can contain the `footnote` element as a child; the other, that expresses that a `para` element contained in `cell` element cannot contain the `footnote` element.

```
<elementRule role='para' label='paraWithFNotes'>
  <mixed>
    <element name='footnote' occurs='*'/>
  </mixed>
</elementRule>
<elementRule role='para' label='paraWithoutFNotes'>
  <mixed>
    <empty/>
  </mixed>
</elementRule>
<tag name='para'/>

<elementRule role='section'>
  <ref label='paraWithFNotes' occurs='*'/>
</elementRule>
<elementRule role='cell'>
  <ref label='paraWithoutFNotes' occurs='*'/>
</elementRule>
```

The same rules can be represented as follows in Schematron:

```
<rule context='E'>
  <report test='parent::parent::cell and parent::para and footnote'>
    Element footnote cannot appear.</report>
</rule>
```

DSD supports this feature using its boolean operators. The usage is similar to the "Elm-Attr" example.

3.1.6 Schema Evolution

An XML schema must be capable of evolving, in order to reflect the changing information requirements and to suitably persist over time. The changes include such situations in which element/attributes are added/deleted or the schema is reorganized [22]. In this section, we consider two important features that are related to schema evolution.

1. *Open model.* An open content model enables the presence of additional elements or attributes within an element. The added items do not need to be declared again.

DTD: No XML-Schema: Yes RELAX: No SOX: No Schematron: Yes DSD: No

In XML-Schema, one can express the openness of element or attribute by using `<any minOccurs='0'`

`maxOccurs='unbounded'/>` or `<anyAttribute minOccurs='0' maxOccurs='unbounded'/>`, respectively. For instance, the following XML-Schema code implies that the element `htmlExample` may contain "any" legal HTML element/attribute keywords under it.

```
<element name="htmlExample">
 <complexType>
  <sequence>
   <any namespace="http://www.w3.org/1999/xhtml"
        minOccurs="1" maxOccurs="unbounded"
        processContents="skip"/>
  </sequence>
  <anyAttribute namespace="http://www.w3.org/1999/xhtml"/>
 </complexType>
</element>
```

In RELAX, it is possible to prevent a specific attribute from being part of some element definition, but it is not possible to extend the property to any attribute (thus simulating the closed model). In Schematron, the content model is open by default. The closed model can be also expressed using a `count()` function in XPath. For instance, the following schema states that the `person` element is closed (when the `name` and `address` are all the sub-elements of the `person`):

```
<rule context='person'>
  <assert test='count(name|address) = count(*)'>
    There is an extra element.</assert>
</rule>
```

Some languages support any-type elements (e.g, `ANY` in DTD), simulating the open-model to some extent, but they require that any-type elements should have child elements that *have been declared*, and we can say that they are not truly "open" in the sense of open-model.

2. *Version.* Sometimes it is desirable to allow several different attributes or element definitions with the same name. This feature states that multiple *versions* of an attribute or element can coexist.

DTD: `No` XML-Schema: `No` RELAX: `Yes` SOX: `No` Schematron: `No` DSD: `Yes`

RELAX or DSD support a notion of non terminal similar to one in context-free grammar and consequently support this feature. In RELAX, for instance, by sharing labels of ⟨ElementRule⟩, it is possible to obtain a similar effect. Suppose an element Section has been defined as follows:

```
<tag name='section'>
<elementRule role='section'>
  <element name='para' occurs='*'/>
</elementRule>
```

Now, suppose that one changed the definition of Section in such a way that it can contain not only Para elements, but also Fig elements:

```
<elementRule role='section'>
  <choice occurs='*'>
    <element name='para'/> <element name='fig'/>
  </choice>
</elementRule>
```

Then, RELAX verifier does not complain about the given XML instance document as long as the instance document matches either of the rule definitions. This is useful for frequently evolving or changing XML schemas. Indeed, the verifier does not have to revise existing rules, but only to check the correctness of new added rules.

XML-Schema has a construct ⟨version⟩ for schema definition, but the current specification does not define any further semantics for that; it is simply provided as a convenience.

3.1.7 Miscellaneous Features

1. *Documentation.* At least, all languages support commenting on schema fragments by using a construct ⟨`!-- comment --`⟩. However, here we consider documentation features beyond commenting such as: textual description to explain a schema fragment for human readers; embedded documentation for application programs, or error/hint messages to aid schema validation and debugging.

DTD: `No` XML-Schema: `Yes` RELAX: `Yes` SOX: `Yes` Schematron: `Yes` DSD: `Yes`

Both XML-Schema and RELAX provide ⟨Documentation⟩ and ⟨Appinfo⟩ elements to support description for both human readers and application programs. SOX provides the ⟨Intro⟩ element to specify an overall introduction to the schema and the ⟨Explain⟩ element to provide a hook for including documentation within a schema fragment. However, there is no support for automatic debugging messages or application programs. In Schematron, by using the assertion semantics provided by constructs ⟨Assert⟩ and ⟨Report⟩, detailed documentation for validating XML structures can be provided. DSD supports three keywords: ⟨Label⟩, ⟨BriefDoc⟩ and ⟨Doc⟩. By using them, it is straightforward to implement, for instance, a debugging system.

2. *Embedded HTML.* Due to HTML popularity, it is often convenient to be able to embed HTML fragments inside XML documents.

DTD: `No` XML-Schema: `Yes` RELAX: `Yes` SOX: `Yes` Schematron: `Partial` DSD: `Yes`

Using the use of ⟨Any⟩ or ⟨AnyAttribute⟩, XML-Schema allows a specification such that any well-formed XML fragments are permissible in a type's content model as long as they use types under the specified namespace. Hence, well-formed HTML codes can be easily embedded in an XML document. The example is shown in Section 3.1.6 (open model). Although DTD also supports a construct `ANY` to be used as a content model of an element, since it does not support namespace, `ANY` cannot contain arbitrary HTML fragments which have not been previously declared in an XML document.

RELAX namespace allows embedding of XHTML elements.SOX provides a similar feature using the ⟨Explain⟩ element. Schematron allows a few HTML tags (e.g., `<p>`, `<emph>`), but not general ones. In DSD, one can use the documentation facility to embed HTML.

3. *Self-describability.* The following languages provide a meta-schema (i.e., representing the schema specification by using the schema definition itself). The meta-schema is useful in bootstrapping the implementation of the language.

DTD: `No` XML-Schema: `Yes` RELAX: `Yes` SOX: `No` Schematron: `Yes` DSD: `Yes`

3.2 Six Query Languages

3.2.1 Data Model

1. *Data model.* The data model of an XML query language formally defines the information contained in the input/output of an XML query processor. The designers of Lorel, XML-QL, XML-GL introduced their own data models. XQL relies on the data modeling features of XML. XSLT extended XPath data model [16]. XQuery relies on the XQuery 1.0 and XPath 2.0 Data Model [29], currently under development at W3C.

Lorel: `Graph` XML-QL: `Graph` XML-GL: `Graph` XSLT: `Tree` XQL: `Tree` XQuery: `Sequence`

In the data model of Lorel, an XML element is a pair of `<eid, value>`, where `eid` is a unique element identifier, and `value` is either an atomic string or a complex value, that in turn contains a list of (1) string-valued `tag`, (2) a (possibly empty) ordered list of pairs of attribute names and atomic values (representing XML attributes), (3) a (possibly empty) ordered list of pairs `<eid, value>` called *crosslink sub-elements* (representing XML IDREF attributes), and (4) a (possibly empty) ordered list of pairs `<eid, value>` called *normal sub-elements* (representing the XML containment relationship). The Lorel data model has

a graph representation. Nodes correspond to the XML data elements, while labeled edges correspond to either *crosslink edges* or *normal edges*. Each XML graph has one or more nodes designated as *entry points*.

In XML-QL, an XML document is modeled by an *XML Graph*; each node is associated with an *object identifier* (OID); edges are labeled with element tag identifiers, intermediate nodes are labeled with sets of attribute-value pairs representing attributes, leaves are labeled with values (e.g., CDATA or PCDATA); each graph has a distinguished node called the *root*.

In XML-GL, the *XML Graphical Data Model (XML-GDM)* is used to represent both DTD and XML documents. In XML-GDM, elements and properties (i.e., attributes, CDATA, PCDATA) are represented as rectangles and circles, respectively. Edges between nodes represent containment or reference relationships. IDREFs are denoted by labeled arcs.

XSLT is based on the tree-based data model of XPath [16] with some additions. In XPath, an XML document is modeled as a tree with seven types of nodes (i.e., root, element, text, attribute, namespace, processing instruction, and comment). For every type of node, there is a way of determining a *string value* that is either part of the node or computed from the string-values of its descendants. Some types of nodes also have an expanded name, which is a pair consisting of a local part and a namespace URI. With respect to XPath data model, some restrictions are relaxed (i.e., the input and result tree may not be a well-formed document, the same URI of the external entities are associated to all XML items contained in those entities, or the XSLT processors may strip the whitespaces of text nodes and freely modify the URI of unparsed entities). Collections of documents can be handled by concatenating them under a root node, resulting in a rooted tree.

XQL designers assume the "XML implied data model", highlighting that a document can be considered an ordered, node-labeled tree, with nodes that represent the document identity, elements, attributes, processing instructions and comments. Semantic relationships between nodes can be hierarchical (parent/child, ancestor/descendant), positional (absolute, relative, range) and sequential (precedes, immediately precedes) [50].

The data model of XQuery relies on XQuery 1.0 and XPath 2.0 Data Model [29], which is a refinement of the tree-based XPath data model [16]. In the XQuery extension, a fragment of a document, or a collection of documents, may lack a common root and may therefore be modeled as a sequence of nodes, each of which may contain nested sequences of nodes. W.r.t. XSLT, it is not necessary to put all the documents under the same root.

The data model introduced in Lorel considers the elements in a given XML document as either ordered or unordered; XML-QL does not introduce an order among elements for the sake of simplicity; in the XML-GL graphical data model a counterclockwise order is imposed by drawing a small trait on the incoming edge of the first displayed element; XQL, XSLT and XQuery data model assume the intrinsic order of XML elements and no order on XML attributes, according to the XML specification [8].

The data models of Lorel, XML-QL, and XML-GL are substantially equivalent; the only significant difference concerns the fact that Lorel data model has a differential management of IDREFS, which is discussed next. Finally, XQL and XSLT data models are a node-labeled tree rather than a graph. XQuery data model is a sequence of nodes (ordered by definition) rather than a plain tree.

2. *Differential management of IDREFs.* XML reserves an attribute of type ID, which assigns a globally unique key to the element. An attribute of type IDREF allows an element to refer to another element with the designed key. Thus, IDREFs are particular strings that can be interpreted as references between elements. With such an interpretation, it is possible to navigate from one element to another; the data model supports *object references*, possibly cyclic. However, if IDREFs are interpreted as strings, then nodes are connected only by containment relationships, and the data model does not support object references. These two interpretations of IDREFs may lead to two different interpretations of the same query. We indicate this property as differential management of IDREFs.

Lorel: Yes XML-QL: No XML-GL: No XSLT: No XQL: No XQuery: Yes

In Lorel, there are two modes of viewing the data model: *semantic* and *literal*. In the semantic mode, the

database is viewed as an interconnected graph; in the literal mode, the database is viewed as an XML tree, and IDREFs are represented as textual strings. Both approaches are important, since, depending on applications, one may wish to process XML data as a literal tree (like the DOM [59]) or as a semantic graph. In the XQuery 1.0 and XPath 2.0 Data Model, adopted by XQuery, the special accessor `id` is used to map an IDREF node (of type `xs:IDREF`) to an ordered sequence of nodes. The Lorel's twofold interpretations are adopted in the XQuery data model as well. This feature simply concerns the data model and does not have any impact on querying IDREFs, as explained in section 3.2.2.3.

3. *Data model compatibility with the related W3C standards.* We compare the languages on the support of the W3C standard and working drafts, covering the XQuery 1.0 and XPath 2.0 data model [29], the XML-Schema [58, 3], and the XQuery 1.0 Formal Semantics (former XML Algebra) [28]. XML-Schema has been largely described earlier in the paper. XQuery working draft is currently the only language to include support of XML-Schema datatypes and some formatting and manipulation operations on them. In XSLT 1.1., no support for XML-Schema is provided. The XQuery 1.0 and XPath 2.0 [29] data model and the XQuery 1.0 Formal Semantics [28] will be expected to provide a precise semantics of the standard XML Query Language and to support physical and logical independence of XML data. As already explained, the XQuery 1.0 and XPath 2.0 data model is a node-labeled, tree-constructor representation, which also includes a concept of node identity to map reference values, like IDREF, XPointer and URI values. The XQuery 1.0 Formal Semantics will serve the need of capturing the semantics of a query language by specifying a set of operations and a set of laws to possibly reduce and optimize query expressions. As happens in database languages, a query could be translated in an algebraic expression, as the first step of a query evaluation engine.

Only XQuery adheres to the definitions of the XQuery 1.0 and XPath 2.0 and of the XQuery 1.0 Formal Semantics. XSLT 1.1 uses a different data model and does not provide an algebra-based semantics. XSLT 2.0 is expected to be complaint to these standards.

Lorel: `No` XML-QL: `No` XML-GL: `No` XSLT: `No` XQL: `No` XQuery: `Yes`

3.2.2 Basic Query Abstractions

1. *Document selection.* Document selection is the result of the application of the query to the document(s), extracting the XML data that satisfy the specified condition.

Lorel: `Yes` XML-QL: `Yes` XML-GL: `Yes` XSLT: `Yes` XQL: `Yes` XQuery: `Yes`

Of course this feature is supported by all the languages. We describe simplified descriptions of the syntax of the six languages. For full details, refer to the corresponding documents. A query in Lorel is structured according to the following, simplified grammar [2] [6]:

```
'select' { select_expr }
[ 'from' { from_expr } ]
[ 'where' { where_expr } ]
```

`Select`, `from`, and `where` expressions, as in OQL, can in turn contain queries. A simplified syntax for defining a query in XML-QL is:

```
Query ::= 'where' { Predicate }
          'construct' { '{' Query '}' }
```

The result specified in the `construct` clause is a piece of XML document, fully specified in terms of tag names and content; content is typically constructed from the object bindings which are determined by the predicate evaluation [26].

An XML-GL query consists of two sets of directed acyclic graphs displayed side by side and separated by a vertical line, where the LHS expresses the query sources (which documents are selected) and predicate

[6] We use a BNF notation where non-terminals are enclosed within quotes, curly brackets denote a list of zero or more elements separated by commas, square brackets denote optionality, the symbol | denotes alternatives.

(which condition must be satisfied), while the RHS represents the construction (which document is produced as result). Explicitly drawn connections or implicit homonims, where unambiguous, indicate the bindings between the LHS and RHS.

In XSLT, a template rule is specified within the `xsl:template` tags. The `match` attribute is a pattern that identifies the source node or nodes to which the rule applies. If omitted, the template rule is matched against all the nodes of the document. The following is the approximate skeleton of a template rule [18]:

```
‘<xsl:template’  [‘match=’ pattern_expr] ‘>’
    {‘<xsl:directive>’}
    {‘<result-elements>’}
‘</xsl:template>’
```

where `pattern_expr` represents an expression written in the XPath language (see section 3.2.3.1 for more details), `result-elements` are the new elements' tags produced in the result and `xsl:directive` is one of the following (for a complete list, refer to [19]):

```
‘<xsl:apply-templates’[‘select=’ pattern_expr]‘>’
‘<xsl:for-each’ ‘select=’ pattern_expr ‘>’
‘<xsl:value-of’ [‘select=’ pattern_expr] ‘>’
‘<xsl:copy-of’   ‘select=’ pattern_expr ‘>’
```

where the `select` attribute is used to process nodes selected by an expression instead of all the children of the current node. The `xsl:apply-templates` directive invokes the application of templates which are separately defined; the `xsl:for-each` directive iterates on other directives which are statically nested within it; the `xsl:value-of` directive extracts information from a pattern expression and converts it into a string; the `xsl:copy-of` directive extracts information from a pattern expression and puts it into the result, with its original format.

The basic XQL syntax mimics the URI directory navigation syntax, but instead of specifying navigation through a physical structure, the navigation is through elements in an XML tree. A simplified syntax for XQL is as follows [53]:

```
Query ::= [ ‘./’ | ‘/’ | ‘//’ | ‘.//’]
          Element [ ‘[’ Predicate ‘]’ ][Path]
Path  ::= [ ‘/’ | ‘//’ ] Element
          [ ‘[’ Predicate ‘]’ ] [Path]
```

Therefore, a query is specified along a hierarchical path within the document; predicates (filters in XQL terminology) typically apply to the elements being accessed during the navigation along the path, although syntactically they may reach elements which are quite distant in the hierarchical structure.

A simplified syntax for defining a query in XQuery is:

```
QueryModuleList ::= QueryModule ( ";" QueryModule)*
QueryModule ::= ContextDecl* FunctionDefn* [ Expression* ]
Expression  ::= Path Expression | ForLetWhereReturnExpression |
                ElementConstructor | SpecialExpression
ForLetWhereReturnExpression ::= ‘for’ { Expression }
                                ‘let’ { Expression }
                                ‘where’ { Predicate }
                                ‘return’ { ‘{‘ Expression ’}’ }
```

The main novelty of XQuery is the definition of a query as a comprehensive combination of query modules. Each query module consists of context declarations (namespace and external schema declarations), function definitions and different kinds of expressions. Admitted expressions are XPath expressions, element constructors, special expressions (such as conditional and computational statements) and the joint use of FLWR (i.e., `FOR`, `LET`, `WHERE`, `RETURN`) clauses. The result specified in a path expression is a piece of XML obtained through a series of steps descending the XML tree; the result in a `return` clause, the result in

an element constructor and the result in a special expression is a piece of XML document, fully specified in terms of tag names, content and binding variables (for full details on XQuery syntax, see [13]).

2. *Full-fledged joins.* Joins combine data from multiple sources into a single result. We analyze which of the reviewed languages supports full-fledged join capabilities (such as intra-documents and inter-documents, inner and outer joins, equi and theta joins).

Lorel: Yes XML-QL: Yes XML-GL: Yes XSLT: Yes XQL: Partial XQuery: Yes

In Lorel, join conditions are fully supported, within the same document and among several documents. They are written in a SQL-like form, by explicitly specifying the variables involved in the joins. In XML-QL, joins are implicitly expressed by means of the equality on variable names, which must match their value. They can range on the same document or on several different documents, defining arbitrary join conditions (e.g., n-way join corresponds to associating the same variable name to n labels in the query). In XML-GL, join conditions are expressed by connecting, by means of a comparison operator, two leaf nodes of XML documents, representing arbitrary attributes or data; equi-joins are represented by edges pointing to the same node, possibly with two labels (in many cases joined nodes have the same label). XQL only partially supports joins because it does not allow to use correlation variables upon the roots of the documents.

XSLT allows intra-document joins by using the `xsl:apply-templates`; each template rule currently addresses a single document and connection conditions can be built within the same document using the `xsl:variable` instruction. The result is a rather complicated nesting of templates. Joins among different documents are admitted using the trick of declaring a variable, which is managed as local within the template, even if linked to an external document. The first version of XQL [53] allowed only semi-joins, i.e., joins of data which is reachable along a path with other data which may be present in the same document. The new extension of XQL [54] allows, as a preliminary feature, joins among different documents by means of correlation variables within the path. For instance, it is possible to ask for all books and all reviews having the same isbn number as the books:

```
book[$i:=isbn] {isbn| title | author | //review[isbn=$i]{reviewer|comment}}
```

If necessary, the function `ref(url)` can be used to refer to a document belonging to a given url. It is still not clear how to join an element which is the root of a document. In such a case, in fact, the filter condition with the variable, cannot be applied to the root.

XQuery fully supports different forms of inner and outer joins on single and multiple documents. The join is obtained either by explicating the relationship of joined data in the filter part (enclosed by square brackets) within the path expression or by putting the join condition inside the `WHERE` clause. For instance, the following snippet shows the case of inner join:

```
FOR $b in document("book.xml")//book,
    $r in document("review.xml")//review[isbn=$b/isbn]
RETURN
    <book>
      $b/isbn,
      $b/title,
      $b/author,
      $r/reviewer,
      $r/comment
    </book>
```

3. *Explicit IDREF dereferencing.* It is particularly noteworthy to distinguish the languages on their specific treatment of IDREFs. The use of an explicit operator or function to dereference IDREFs allows to write more compact and intuitive expressions and avoids an explicit join between the two linked elements. This operator can be considered as the counterpart of the referencing operator for pointers in traditional programming languages.

Lorel: Yes XML-QL: No XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

In Lorel, no explicit dereferencing operator is present but IDREFs can be transparently crossed such as

normal sub-elements using the dot notation. In XML-QL, an explicit join must be added to express a dereferencing on an IDREF attribute. In XML-GL, a labeled arc is used to represent the reference. In both XQL and XSLT, `id()` function is provided for dereferencing, while in XQuery, `=>` is introduced to follow an attribute with `IDREF` type or a foreign key.

4. *Multiple output of the query processor.* A query processor operates on XML data by actually mapping them into the information structures provided by the underlying data model. Thereafter, the result, represented as a data model instance, is again translated into a new XML document, which can be then queried and possibly updated independently. Only XSLT and XQuery supports as output multiple documents.

Lorel: No XML-QL: No XML-GL: No XSLT: Yes XQL: No XQuery: Yes

Lorel, XML-QL and XML-GL features the query result in terms of a single output graph and then of a single XML document. XQL processor produces as output a single tree data model instance and then a single document. XSLT 1.1 has added the multiple output feature through the `xsl:document` directive. Finally, XQuery uses the XQuery 1.0 and XPath 2.0 data model, thus possibly yielding as output collections of XML documents.

5. *Definition of views.* In conventional databases, any relation that is not part of the logical model but is made visible to a user as a virtual relation is called a *view*. Similarly, in XML model, the result of a query can be either XML data fragments or a view on the document base.

Lorel: Yes XML-QL: No XML-GL: No XSLT: No XQL: No XQuery: No

In Lorel, the result of a query is a set of object identifiers pointed by a new element. Therefore, the standard interpretation is that the current state of objects being selected is the one present in the database, and subsequent accesses to the query result may give different documents. In Lorel it is also possible to define views (`with` clause), and in such case the query returns a document with all the nodes which are specified by the `with` clause. In the standard interpretations, XML-QL, XML-GL, XSLT, XQL and XQuery return new documents, whose content is then independent on subsequent database manipulations. None of these languages currently supports the definition of views.

3.2.3 Path expressions

1. *Partially specified path expressions.* When querying semistructured data, especially when the exact structure is not known, it is convenient to use a form of "navigational" query based on path expressions. The most powerful form of path expression does not need to list all the elements of the path, as it uses wildcards and regular expressions: we denote it as a *partially specified path expression*. All languages support partially specified path expression and consider this feature as one of the most important in the language.

Lorel: Yes XML-QL: Yes XML-GL: Partial XSLT: Yes XQL: Yes XQuery: Yes

Lorel supports path expressions with several UNIX-like wildcards. Each path expression must have a context (the root element of the document). In XML-QL, path expressions are admitted within the tag specification and they permit the alternation, concatenation and Kleene-star operators, similar to those used in regular expressions. XML-QL path expressions have the same expressive power as those of Lorel. In XML-GL, the only path expressions supported are arbitrary containment, by means of a wildcard * as edge label; this allows traversing XML-GL graph reaching an element at any level of depth.

In XSLT, XQuery (XPath language) and XQL, path expressions define relative and absolute locations. A relative location path consists of one or more location steps separated by the child '/' operator or the descendant '//' operator. Each step selects a set of nodes relative to a context node. An absolute location path has a '/' or '//', optionally followed by a relative location path. The final set of nodes of either a relative or absolute location path is the union of the set of nodes of each step. Admitted wildcards are both alternation and star operator. The star operator does not enforce the closure which is instead

obtained by the descendant '//' operator. Indeed, the star operator simply stands for retaining in the result any element.

2. *Matching of partially specified expressions with cyclic data.* Partially specified expressions may be the source of infinite computations in the case of cyclic instances. Therefore, it is common practice to specify halt conditions in the matching algorithm that binds object instances to path expressions when the same object binding is associated to the same query node more than once.

Lorel: Yes XML-QL: No XML-GL: Yes XSLT: No XQL: No XQuery: No

Some systems (Lorel and XML-GL) mention the halt condition as part of the query language semantics. In both Lorel and XML-GL, the same node cannot be visited more than once. By contrast, we expect that the other languages (XML-QL,XSLT,XQL and XQuery) include this feature in their implementations. For instance, in XSLT `xsl:apply-templates` are used to process only nodes that are descendants of the current node, and this cannot result in non-terminating processing loops. However, non-terminating loops may arise when `xsl:apply-templates` is used to process elements that are not descendant of the current node. For example, the template rule:

```
<xsl:template   match="foo">
  <xsl:apply-templates    select="."/>
</xsl:template>
```

matches the `<foo>` elements at all levels of nesting, including the level on which the matching occurs, yielding to a possible infinite call sequence.

3.2.4 Quantification, Negation, Reduction and Filtering

1. *Existential quantification.* An existential predicate over a set of instances (e.g., bound to a variable) is satisfied if at least one of the instances satisfies the predicate.

Lorel: Yes XML-QL: Yes XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

In all languages except XQuery, predicates are assumed as existentially quantified. In XQuery, when it is necessary to test for existence of some element that satisfies a condition, the special expression "`some <quantifier_variable> in <expression> satisfies <value>`" is used in the `WHERE` clause.

2. *Universal quantification.* An universal predicate over a set of instances (e.g., bound to a variable) is satisfied if all the instances satisfy the predicate.

Lorel: Yes XML-QL: No XML-GL: No XSLT: No XQL: Yes XQuery: Yes

In Lorel, a variable can be universally quantified with the SQL-like predicate `for all`. Similarly, in XQL universal quantification is obtained by prefixing a predicate expression with the keyword $*all*$. In XML-QL and XML-GL[7], universal quantification cannot be expressed. In XSLT, conditions cannot be universally quantified. In XQuery, an additional special expression "`every <quantifier_variable> in <expression> satisfies <value>`" has been introduced to express universal quantification.

3. *Negation.* A negated predicate over a set of instances is satisfied if none of the instances satisfies the same predicate without considering negation.

Lorel: Yes XML-QL: No XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

In Lorel, a predicate is negated with the key-word `not`. In XML-GL, the negation is expressed graphically by a dashed edge (see Figure 1, where all professors without name elements are selected). In XQL, a function *not*() is provided to negate the value of an expression:

```
professor[not(name)]
```

[7]However, in XML-GL, the presence of negation and existential quantification allows the expression of universal quantification by means of views.

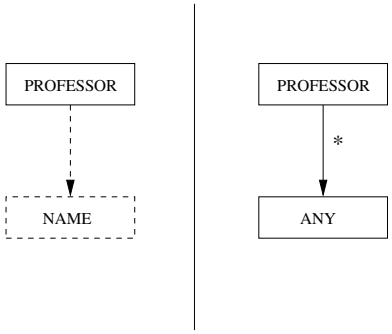

Figure 1: Example of negation in XML-GL

Its meaning is to select all the instances of the context where the positive expression (e.g. having a `name`) is false or not valuable (e.g., the name sub-element is missing). XML-QL does not support negation (it supports the unequal comparison operator in simple predicate expressions). In XSLT pattern language, negation is expressed by means of the boolean function `not` that operates on a boolean value and returns a boolean (before using the `not` function, a number, a string, a node set and an object must be first converted to their boolean value by applying the `boolean` function to them). In XQuery, the negation is expressed using the function `not`, that takes a boolean value as its argument and returns the logical negation of the argument. Alternatively, a built-in function *empty* is used to query the absence of tags.

```
for $p in //professor
where empty($p/name)
return $p
```

4. *Reduction.* Given a document and a query on this document, the reduction prunes from the document those elements specified in the selection part of the query that satisfy its condition.

Lorel: `No` XML-QL: `No` XML-GL: `No` XSLT: `Yes` XQL: `No` XQuery: `No`

XSLT is the only language in which reduction is possible by writing one template for each element to be dropped and by leaving these templates empty. In Lorel, XML-QL, XML-GL and XQuery, document reduction is not supported, also if it could be obtained through a construction which includes only the elements that should remain. In XQL no construction can be expressed, and therefore this approach is not feasible.

5. *Filtering.* Filtering means retaining from the documents the specified elements while preserving hierarchy and sequence. It can be obtained through several nested select/construct clauses, but the availability of an ad-hoc operator can help the readability of the query.

Lorel: `No` XML-QL: `No` XML-GL: `No` XSLT: `Partial` XQL: `No` XQuery: `Yes`

In XQuery, an explicit *Filter* function has been defined. It takes one operand, which can be any expression. The function evaluates its argument and returns a shallow copy of the nodes that are selected by the argument, preserving any relationships that exist among these nodes. In XSLT, the filtering operation is obtained using a set of templates, and precisely a template per each element to be retained. The structure of the document is preserved, and detached templates turn out less complicated than nested queries with clauses. But filtering is only partially supported by XSLT, since neither an explicit operator nor a built-in function has been defined.

3.2.5 Restructuring abstractions

1. *Building new XML data.* New XML data can be created through the query's construction mechanism.

Lorel: `Yes` XML-QL: `Yes` XML-GL: `Yes` XSLT: `Yes` XQL: `No` XQuery: `Yes`

In Lorel, a new XML item is built by invoking the `xml()` function with three parameters (the first two

not mandatory): the type, the label and the value(s), explicitly given through the OID or implicitly given specifying the query that generate it.

In XML-QL, restructuring is specified in the `Construct` clause, which contains the new tags, text, and binding variables (bound in the XML-QL predicate evaluation) of the new document, arbitrarily named. In XML-GL, new XML items are constructed by drawing in the right-hand side of the query arbitrarily named graph symbols. In XSLT, the new components of the query result are specified through their tag names within the template rule or generated by specific xsl directives.

In XQL, new items cannot be added to the existing ones, because no construction mechanism is provided. In the new version of XQL [54], a renaming operator, represented by an arrow, has been defined, but it can only rename a tag, which is supposed to be already existing. In XQuery, the new items are specified through their tag names in the `RETURN` clause (invoking the element constructors) or suitably generated by functions.

2. *Grouping.* Elements of the result can be aggregated or reorganized as specified by means of special functions, such as `group by`.

Lorel: `Yes` XML-QL: `No` XML-GL: `Yes` XSLT: `Partial` XQL: `Yes` XQuery: `Yes`

In Lorel, the `group by` clause is inherited from the OQL. Objects extracted in XML-GL can be grouped according to the distinct values of one of their attributes or PCDATA; each class is associated to an element instance carrying the representative value of the class and then the rest of the XML tree as sub-element. Apparently, an explicit group-by clause is missing from the current descriptions of XML-QL, XSLT and XQL. However, in XQL, the use of curly brackets makes it possible the group the results. For instance, the following code groups products by invoice, placing each group of products within an invoice tag [54]:

```
//invoice { . //product }
```

In XSLT, it is possible to simulate grouping by suitably nesting `xsl:apply-templates`, parameterized templates and `following-siblings` axis. XSLT code that implements the grouping is verbose and poorly readable. In XQuery, the grouping function is implemented by `LET` clause, which binds to a set of tuples.

3. *OID invention with a Skolem function.* A Skolem function generates a unique OID to a given parameter and are useful in integrating existing documents into one.

Lorel: `Yes` XML-QL: `Yes` XML-GL: `Partial` XSLT: `No` XQL: `No` XQuery: `Yes`

In Lorel, XML-QL, XML-GL and XQuery, the Skolem function takes as input a list of variables and returns one unique element for every binding of elements and/or attributes to the argument. In Lorel and XML-QL, the Skolem function associates to each element a unique object identifier, that can be referenced by variables; XML-GL, instead, does not provide the possibility of explicitly referencing the object identifiers. In XSLT and XQL, the Skolem functions are not supported.

3.2.6 Aggregation, Nesting, and Set operations

1. *Aggregates.* Aggregate functions compute a scalar value out of a multi-set of values. Classical aggregates, supported by SQL, are `min`, `max`, `sum`, `count`, `avg`.

Lorel: `Yes` XML-QL: `No` XML-GL: `Yes` XSLT: `Partial` XQL: `Partial` XQuery: `Yes`

In Lorel and XQuery, the aggregate functions are present and fully implemented. In XML-QL, as the time of writing, they are not supported, but indicated to be supported in the next version. In XML-GL, aggregates are represented graphically. Both XSLT and XQL support `count` and `sum` functions, but not

others.

2. *Nesting of queries.* As in SQL, a query can be composed of nested subqueries.

Lorel: Yes XML-QL: Yes XML-GL: No XSLT: Yes XQL: Yes XQuery: Yes

In Lorel and XML-QL, both inspired by SQL paradigm, queries can be nested at an arbitrary level. In XML-GL, nesting is not supported. In XSLT, templates can be nested using the xsl command `xsl:apply-templates` or `xsl:call-template`. In XQL, queries can be nested within the square and curly brackets. In XQuery, the different kinds of expressions, which embody queries, can be arbitrarily nested.

3. *Set operations.* As in SQL, a query can be binary, composed of the union, intersection, or difference of subqueries.

Lorel: Yes XML-QL: Partial XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

Both Lorel and XQuery support union, intersection, and difference operator. XML-QL supports union and intersection but has no difference. In XML-GL, queries allow multiple graphs in the left side of the query; this gives the expressive power of union. Negation gives to XML-GL the expressive power of a difference, and intersection can be built by repeatedly applying negation. XSLT admits union and negation in the pattern language; there is no explicit intersection. XQL supports union and intersection; negation gives to XQL the expressive power of a difference.

3.2.7 Order Management

1. *Ordering the result.* Consists of ordering the element instances according to the ascending or descending values of some data of the result, as performed by the `order by` clause in SQL.

Lorel: Yes XML-QL: Yes XML-GL: Yes XSLT: Yes XQL: No XQuery: Yes

In both Lorel and XML-QL, ordering of the result is gained through the `order by` clause. In XML-GL, a leaf node can be labeled with `ASC`, `DESC ORDER` and the elements extracted are in ascending or descending order with respect to that node; multiple labelings are possible. In XSLT, sorting is specified by adding `xsl:sort` elements as children of `xsl:apply-templates` or `xsl:for-each`. Each `xsl:sort` specifies a sort key: the elements are ordered according to the specified sort keys, instead of being ordered on the basis of the document order (default). In XQL, (as from available documentation) there are no clauses concerning the order of the result. In XQuery, the `sortby` clause is introduced and may be used after an element constructor or path expression to specify an ordering, ascending and descending, among the resulting elements.

2. *Order-preserving result.* This features indicates if the ordering the elements in the result can be preserved in the same way as the original document.

Lorel: Yes XML-QL: Yes XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

In Lorel, the `order by document order` means that the retrieved elements are ordered in the same way as the original document, and the new ones are placed at the end of the document with an unspecified order among them; otherwise, the order of elements in the result is left unspecified. XML-QL, XSLT and XQL produce ordered XML, therefore the order can be specified to be the same as the source document (but such an order must be known through the schema). XML-GL produces an ordered XML element when one of the edges is marked (then, the elements are produced in counterclockwise order with respect to the marked edge). An XQuery query preserves the ordering of elements in the input documents, as represented by the values of its bound variables. If some expression in a `FOR` or `LET` clause is unordered, the tuples generated by these clauses are unordered too.

3. *Querying the order of elements.* Consists of asking for XML elements in a given order relationship,

i.e., before or after other specified ones.

Lorel: No XML-QL: Yes XML-GL: Yes XSLT: Yes XQL: Yes XQuery: Yes

All languages, except Lorel, support this feature. In XML-QL, it is possible to query about the relative position of tags within documents (asking if a tag precedes or follows another tag) using tag variables. In XML-GL, the order of elements is superimposed by adding a small trait on the incoming edge of the first element and the siblings are intended in a counterclockwise order. In both XQuery and XQL, the keywords `BEFORE` and `AFTER` have been introduced to query the order of elements. This feature is not supported by the other languages. In XSLT, it is possible to use the `preceding` and `following` axis to extract the elements that respectively precede or follow the context node.

4. *Querying numbered instances.* This feature indicates a capability of asking for numbered instances of XML elements. This is accomplished adding to the language the range qualifier operator, that enables the selection of a single number (or of a set of numbers) instances.

Lorel: Yes XML-QL: No XML-GL: No XSLT: Yes XQL: Yes XQuery: Yes

In Lorel, the operator "[<range>]" is introduced into both the query and update language and it is applied against both a path expression or variable. In XML-QL and XML-GL, there is no range selection. In XSLT pattern language and in XQL, a specific node within a set of nodes is extracted simply enclosing the index ordinal within square brackets in the pattern. XQuery also allows, inside the square brackets, a sequence of literal numbers, and a range expression that generates a sequence of consecutive integers.

3.2.8 Typing & Extensibility

1. *Support of datatypes.* This feature concerns the necessity of embedding in an XML query language specialized types, inspired by the type system of programming languages. This feature becomes especially relevant as XML-Schema will enhance the initial XML definition by adding different primitive and defined types (refer to 3.1.2). Within the query language, these types might be used in function definitions, query conditions or type manipulation operators.

Lorel: Partial XML-QL: No XML-GL: No XSLT: No XQL: No XQuery: Yes

Lorel supports only some specialized data types such as audio, video, images, JPEG, GIF, and PS. Support of XML-Schema datatypes appears as a must requirement in the XSLT 2.0 requirements, but it is not supported by the current recommendation and working draft [18, 19]. The type system of XQuery is that of XML-Schema. In XQuery, types appear in function declaration, in operators tailored to type coercion (see next sub-section), and possibly, in future extensions, more complex user-defined datatypes might be declared in XML Schema notation in a preamble of the queries.

2. *Type coercion.* This feature concerns the ability of implicit or explicit data casting among different types, as well as the ability to compare values represented with different type constructors (e.g., scalars, singleton sets, and lists with one only element). Because of the nature of semistructured data, the type coercion provided by an XML query language should be much more flexible than that of a database query language.

Lorel: Yes XML-QL: No XML-GL: No XSLT: No XQL: Yes XQuery: Yes

In Lorel, comparison between objects and/or values is forced to follow the most common intuition when comparing objects and values of different types. Coercion rules are provided for the various atomic types and the corresponding predicates or functions. Furthermore, a comparison between atomic objects, complex objects, and sets of objects is accepted when there is an obvious interpretation. In XQL, two values are comparable only after explicit casting of them. as it happens in a traditional programming language. In XSLT, ad-hoc type coercion operators are promised as a future development. XQuery provides a boolean operator `INSTANCEOF` (that returns true if the first operand is an instance of the type named in the second operand), a `CAST` notation (that accrues the conversion between certain admitted

combination of types), and a `TREAT` notation (that tells the query processor not to do the conversion, but to treat an expression as though it were of a subtype of its static type). Moreover, the `TYPESWITCH` keyword serves the need of dynamically assigning the dynamic type of a generic expression being tested (named typeswitch expression). The other languages do not mention type coercion.

3. *Support of functions.* The availability of a core function library, collecting built-in or user-defined functions, is a natural extension of an XML query language to define special operations on XML data, which are not feasible by means of queries. Moreover, it should be possible to use extension functions, possibly defined in external programming languages. We will examine separately the support of built-in, user-defined and extension functions for all the analysed languages. None of the language fully covers all of them.

Lorel: No XML-QL: Partial XML-GL: No XSLT: Partial XQL: Partial XQuery: Partial

In XML-QL, the keyword `FUNCTION` allows the definition of functions, whose body contains an XML-QL query. Built-in functions, such as those available in XPath definition, are not embedded in the language. Extension functions are not admitted. In XSLT, XQL and XQuery, the built-in functions provided by XPath definition, are available. These functions are basically splitted into four classes: node set functions to implement operations on nodes; string functions to manipulate strings; boolean functions to return boolean values and number functions to do the conversion of number types. XSLT has an additional powerful mechanism to define extension functions, using the `xsl:script` directive. This directive provides the implementation of external functions in a particular namespace. However, although XSLT templates may emulate some function capabilities, they always operates on trees, and are not powerful enough for writing user-definable functions, with arbitrary results (node-sets or values). Moreover, XSLT lacks some XQuery functions, such as `distinct()` and certain aggregation functions 3.2.6. XQuery provides its own function library, which contains all the functions of the XPath core function library, the aggregation functions of SQL and other XQuery functions, such as `empty()` and `distinct()`. In addition, XQuery allows users to define functions of their own using the construct `"DEFINE" "FUNCTION" QName "(" ParamList? ")" ("RETURNS" Datatype)? EnclosedExpr`. The current version of XQuery lacks an extensibility mechanism to define functions in external programming languages. Further study deserves also the subject of function overloading and polymorphic functions.

3.2.9 Integration with XML

1. *Support of RDF.* RDF [38, 10] is an emerging standard for representing metadata for XML documents, and will provide an ontology system for the Web. It may be desirable that an XML query language be compatible with RDF language. At the time of writing, none of the six languages support RDF.

Lorel: No XML-QL: No XML-GL: No XSLT: No XQL: No XQuery: No

2. *Support of XPointer and XLink.* XLink [25] is a W3C proposed recommendation for a language to express linking structures similar to hyperlinks in hypermedia systems. It uses URI to enhance traditional linking technology with more flexible rendering and processing of linked resources. Furthermore, it exploits the self-describing XML syntax to express links both for human readers and for computer programs. XPointer [24] is built on top of XPath to allow addressing the internal structure of an XML document, such as text, any XML data, including external parsed entities. XLink and XPointer could influence the evolution of an XML query language; a query should be able to manipulate special kinds of XML data, such as XML linking elements (XLink-conformant XML elements, that assert the existence of a link) and fragment identifiers (XPointer-conformant resources, identified by URIs). None of the languages currently supports manipulation of these data. At the time of writing, none of the six languages

support XPointer and XLink.

Lorel: No XML-QL: No XML-GL: No XSLT: No XQL: No XQuery: No

3. *Support of namespaces.* An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names.

Lorel: No XML-QL: Partial XML-GL: No XSLT: Yes XQL: Yes XQuery: Yes

In XML-QL, it is possible to qualify tag names with xml namespace prefixes; however, it is not possible to declare them. In XQL, a namespace can be declared using a variable declaration, and used in the query. In XSLT, a namespace declaration can be done using the directive `xmlns:` of xml namespaces within the tag name. Namespaces are next used prefixing them to the tag name. XQuery supports both namespace declaration and usage. Declaration of namespaces is obtained with the keyword `NAMESPACE`.

4. *Support of tag variables.* This feature concerns the possibility of explicitly querying the tag name rather than the tag content.

Lorel: Yes XML-QL: Yes XML-GL: No XSLT: Yes XQL: No XQuery: Yes

In Lorel, path expressions return into variables of a special *path* type the desired tag names to be retrieved. Such names can be used for building the query result using the `unquote()` function. In both XML-QL and XQuery, variables can be assigned to tags and then used to generate the tags of the result. In XSLT, tag variables can be associated with a tag, by using the `xsl:element` directive.

3.2.10 Update language

1. *Support for insert, delete, and update.* Being able to update XML elements/attributes plays an important role in maintaining XML document in the long run.

Lorel: Yes XML-QL: No XML-GL: Yes XSLT: Yes XQL: No XQuery: No

Lorel has an `update` construct so that it is possible to create and delete object names, create an atomic or complex object, and modify the value of an existing object. Update operations in XML-GL are graphically indicated as arrows labelled with `I, D, U`. Each primitive (except for delete that does not need the RHS) has a LHS element and a RHS graph; LHS is the target element of the operation and RHS graph represents the values to be inserted or replaced by the primitive. Being a transformation language, XSLT can simulate the primitives of an update language by using templates for dropping elements, inserting new content, and transforming the existing XML data. The remaining languages do not support update yet.

4 Desired Qualities of the Schema and Query Languages

Besides the comparative review of features, summarized in Tables 1 and 2, we will briefly discuss the desired qualities of schema and query languages, and propose a relative ranking of them, by evaluating the feature coverage.

1. *Ease of use.* This property indicates how easy it is to write and/or to understand a schema and a query, respectively from a schema designer or a query programmer perspective.

   Despite its proprietary syntax, DTD is arguably the easiest schema language to learn and use. Since the new additions to RELAX and SOX are relatively manageable, we think the migration curve from DTD to these languages is not steep. Although the language specification of Schematron is simple, the users are required to learn yet another language, XPath. Due to their extensive set of features, we expect XML-Schema and DSD to be more difficult to learn than the other languages. Since DSD

uses explicit operators for regular expressions (e.g., ⟨Repeat⟩, ⟨OneOf⟩), a DSD schema tends to be more verbose than an XML-Schema or a Schematron schema and, thus, more difficult to decipher.

For what concerns query languages, XML-GL is easy to learn and to use, as it is associated to a graphic interface, while XQL is also simple and easy, but much less powerful; these two languages are the easiest to use. With respect to XQL, XSLT is less readable because of the instruction tags and of the template rules. XQuery, Lorel, XML-QL are comparable with respect to the ease of formulation, although database experts will be probably more familiar with Lorel style, and XML experts will probably be more familiar with XQuery and XML-QL. One should also notice that XML-QL queries are rather verbose if compared with Lorel queries. XQuery queries are less verbose than XML-QL queries, but are still not comparable with the compact XQL expressions.

2. *Database support.* To structure and inspect a broad spectrum of information sources, XML languages should be versatile enough to embed database-related features. Moreover, experience accumulated within database management systems could suggest advanced database features and can offer suitable guidelines for the definition of both Schema and Query languages.

   From this perspective, no single language suffices the needs completely. The DDL of SQL not only allows the specification of a set of relations and attributes, but also the definition of the domain of values associated with each attribute, integrity constraints, indexes for each relation, the mapping of schema types into host programming languages, the authorization for roles and priviledges, and other security mechanisms etc [57]. While XML-Schema fulfills the support for a variety of built-in domain types, it could not express, for instance, an arbitrary SQL `CHECK` or `ASSERT` clause (e.g., sum of columns A and B should be less than 100). Furthermore, although Schematron or DSD can express such integrity constraints, they have no support of physical indexes specification for boosting performance. Since a substantial amount of web documents are generated from underlying databases by user's requests, it is important to be able to handle such data-centric features as SQL DDL do.

   Among the features that are supported by DML of SQL and are relevant for XML Query languages, there are special operators for OLAP analysis and additional aggregates, transaction-related, diagnostics-related and connection-related statements, support of triggers, and programming language extensibility, i.e. the use of external functions and operators. Currently none of these feature is handled by query languages. XQuery is expected to move towards the enhancement of function definition, but we feel that this is one of the areas where database researchers can contribute most in the future.

3. *Expressive power.* This indicates how powerful a language is in expressing schemas and queries. We have drawn separate analysis for Schema and Query languages. The evaluation of expressiveness pertains to the coverage that each of the languages has gained over the features. By considering the number of covered features, we have obtained a *relative ranking* of the expressive power of languages.

   First, when we look at the "intrinsec nature" of the language, the six reviewed XML schema languages can be roughly divided into two groups based on factors such as grammar-based vs. pattern-based, definition-oriented vs. validation-oriented, structure-oriented vs. constraints-oriented, etc [41]. Based on our study, DTD, XML-Schema, RELAX and SOX belong to the grammar-based (or equivalently definition-oriented or structure-oriented) language group while Schematron belongs to the pattern-based (or equivalently validation-oriented or constraints-oriented) language group. DSD stands in between, supporting both features together. The grammar-based language group greatly facilitates the querying task, because knowing the structure and definition of the schema helps users to optimize queries and detect possible errors. On the other hand, the pattern-based language group is naturally superior with respect to the expressiveness of schema constraints.

   Second, when we look at the individual "features" supported by the language, XML-Schema is the most expressive schema language; it supports most of the features that we inspected except a

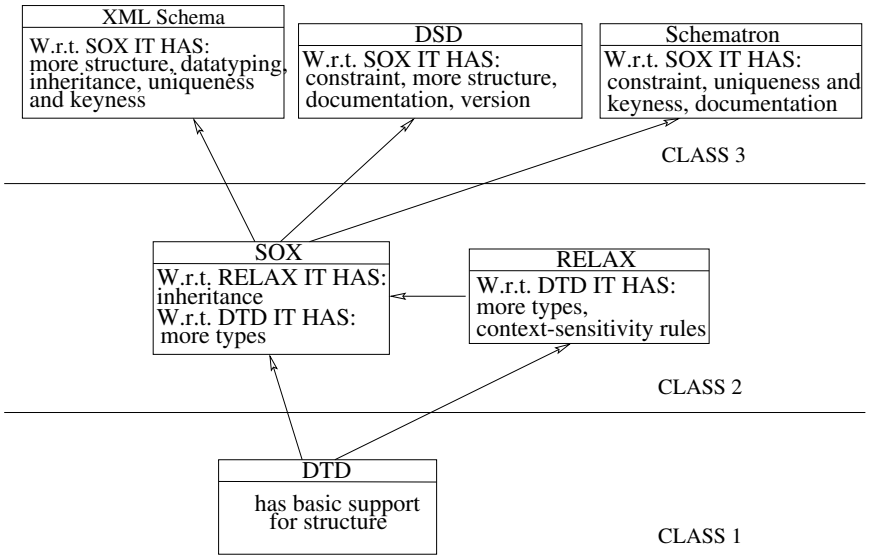

Figure 2: Classification of the expressive power of the six schema languages.

few features related to constraints. Compared to XML-Schema, DSD lacks namespaces, datatypes, OO features, and key-related features. Similarly, with respect to XML-Schema, Schematron lacks datatypes, modularity, and OO features. However, both languages are better in their support of context-sensitivity than XML-Schema as exhibited in Section 3.1.5. The support of datatypes in SOX is not as extensive as that in XML-Schema. In addition, compared to DSD and Schematron, SOX lacks the capabilities of domain constraints, context-sensitive definition, content model, unique and key properties, version control, and self-describability. Although SOX claims a full support of OO features, its support of OO capabilities are behind that of XML-Schema. RELAX excels in its support of context-sensitive rule definition, but lacks the support of inheritance, unique and key properties, and default value with respect to XML-Schema, DSD, and Schematron. DTD supports only the basic datatypes and content model and thus is likely to be the less expressive language among the six.

A visualization of the expressive power of schema languages is given in Figure 2. Now let us look at the expressive power of query languages.

XQuery is the most powerful language. According to the features that we have listed, XQuery is powerful for what concerns the data model, filtering, grouping, complete order management, and datatypes; some remaining features are lacking and need to be incorporated, such as reduction, definition of views, embedding of RDF, XPointer and XLink, and an update language. Lorel is less powerful than XQuery, since it lacks among the other features a complete order management, the support of functions and filtering. However, it has the differential management of IDREFs within the data model, and an update language. XSLT is a stylesheet language which has been empowered with full-fledged query capabilities. In particular, by using variables, it is possible to make joins among different documents; by combining templates, it is possible to implement the reduction feature. With respect to Lorel and XQuery, it lacks universal quantification, aggregates, Skolem functions, type management, tag variables and an update language. It has only partially defined filtering, grouping and aggregates.

XML-GL is less powerful than both XQuery and Lorel, mainly because of the limitations in supporting nested queries, certain specified path expressions, universal quantification and Skolem functions. Due to its graphical nature, the language supports what is naturally specified in a graphical formalism, and can be considered as a QBE-equivalent of the other languages. Indeed, QBE is less expressive than SQL in the relational world, however capturing the essential expressive power of relational languages. XML-QL can be considered the predecessor of XQuery, and it has an expressive

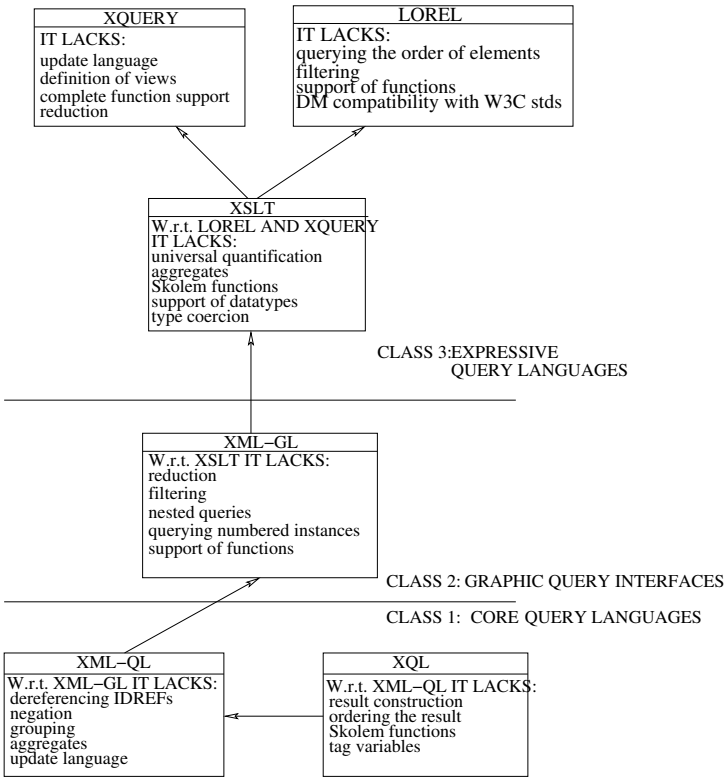

Figure 3: Classification of the expressive power of the six query languages. (None of the six languages supports: RDF embedding, XPointer and XLink embedding. Indeed, reduction is supported by XSLT only.)

power that can be considered equal to the new XQL. With respect to XML-GL, it lacks among the other features an explicit operator for dereferencing IDREFs, negation, grouping, aggregates, and an update language.

XQL has been enhanced with respect to the previously published version of the language, in order to support partial joins, links and functions. However, with respect to XML-QL, it still lacks the capability of constructing new elements/documents, the ordering of results, Skolem functions, and tag variables.

XQuery is a very powerful language explicitly designed for managing XML data. Lorel was defined for semi-structured data, but anticipated many relevant features of XML query languages. XSLT is a stylesheet language enabling a document query language with transformation capabilities. It is currently the most widely used language, due to the availability of many XSL processors.

A visualization of the expressive power of query languages is given in Figure 3.

5 Taxonomy

In this section, we present a taxonomy of XML schema and query languages, where they are assigned to distinct classes: the languages belonging to the same class exhibit only trivial differences and are

themselves involved in a hierarchical relationship. Moreover, the classification has been made, taking into account the number of supported, partial and lacking features as in the Tables 1 and 2. First, the six XML schema languages can be organized into the following three classes as depicted in Figure 2.

1. **Class 1**: DTD is representative of **Class 1 of core schema language for XML**, with a restricted expressive power. It does minimally support the basic schema abstractions and severely lacks datatypes and constraints definition. Since the expressive power of DTD is strictly weaker than other schema languages, the translation from other schema languages to DTD is straighforward while the reverse translation is likely to be lossy.

2. **Class 2**: RELAX and SOX belongs to the middle tier and can be considered representatives of **Class 2 of extended schema language for XML**. Their support for datatypes is not enough (e.g., lack of explicit null and user-defined type) although basic schema abstractions can be supported rather sufficiently. In addition, they lack the support for the full range of content models and database-oriented features such as uniqueness and keyness. Like DTD they mostly fail to fully handle constraint specifications.

3. **Class 3**: XML-Schema, Schematron and DSD are the most expressive languages and are representatives of **Class 3 of expressive schema languages for XML**. Whereas XML-Schema fully supports features for schema datatype and structure, Schematron provides a flexible pattern language that can describe the detailed semantics of the schema. DSD tries to support common features supported by XML-Schema (e.g, structure) and Schematron (e.g., constraint) along with some additional features.

In our study, we have found that the support of constraints in the schema language (e.g., Schematron, DSD) is a very attractive feature. At the same time, ignoring the schema definition aspect completely, as it happens in Schematron, raises some concerns about the intrinsec complexity of a general purpose schema language. Although XML-Schema identifies many commonly recurring schema constraints and incorporates them into the language specification, we still feel that XML-Schema is not flexible enough. It would be interesting to see whether support of new constraints will be added to the new versions of XML-Schema.

The six XML query languages can be organized into the following three classes as depicted in Figure 3. Finally note that this taxonomy is slightly different from that presented in our previous work [4], due to the recent changes to some query languages.

1. **Class 1**: The new XQL and XML-QL are representative of **Class 1 of core query languages for XML**, playing the same role as core SQL standards and languages (e.g., the SQL supported by ODBC) in the relational world. Their expressive power is included within the expressive power of XSLT.

2. **Class 2**: XML-GL can be considered representative of **Class 2 of graphical query interfaces to XML**, playing the same role as graphical query interfaces (e.g., QBE) in the relational world. The queries being supported by XML-GL are the most relevant queries supported by XSLT. It can suitably be adopted as a front-end to any of these query languages (more or less powerful), to express a comprehensive class of queries (a subset of them in case of more powerful languages).

3. **Class 3**: XQuery, Lorel and XSLT are representative of **Class 3 of expressive query languages for XML**, playing the same role as high-level SQL standards and languages (e.g., SQL2) in the relational world. XQuery and Lorel are quite different in their syntax and semantics, due to their completely distinct nature (the semi-structured approach of Lorel as opposed to the XML-inspired mainstream of XQuery). Moreover, Lorel is strongly object-oriented, while XQuery can be considered value-oriented. XQuery is a promising expressive query language, that realizes its potentiality by incorporating the experience of XPath and XQL on one side, of SQL, OQL and XML-QL on the other side. The third language of this class XSLT covers a lower position in the taxonomy being

less powerful than the previous two. It is a stylesheet language with a fairly procedural tendency, as opposed to Lorel, which can be considered completely declarative, and to XQuery, which blends a declarative and procedural flavor.

XML languages standardization activities are in their infancy, especially if compared with the historical SQL development. We believe that, before committing these activities, many features are still to be understood. We have identified features that are commonly inspired by the languages missions and by the past database practitioner's experience. Nevertheless, we do not exclude that, when the new breed of information systems will pervade the Web, additional or different features might come up.

6 Conclusions

In this paper we have presented a comprehensive analysis of a dozen XML schema and query languages. Our comparative review of their features are summarized in Tables 1 and 2, respectively. Comparative examples of those schema and query languages for real-life scenarios greatly helped us to understand the languages and to realize the differences among them. We have done such comparisons, but omitted them here due to space limitations. Please refer to schema language examples[8] and query language examples[9] for further details.

As a general comment, we have noticed, however, that the philosophies by which each language has been designed are quite different; some of them try to define more semantics while others try to be more minimalistic. Therefore, the languages with more features should not be necessarily regarded as superior to those with less features. The goals of our work are twofold. First, we hope the materials presented here are used as checkpoints before people decide which schema or query languages to use in their applications. Given a vast number of options available, choosing the right schema or query language for an application is a daunting yet crucial task. Second, our work has been done aiming to support the standardization process, currently ongoing at W3C and OASIS. This process is expected to yield soon a standard schema and query languages for XML data.

Acknowledgments

Angela Bonifati is particularly grateful to her PhD advisor, Prof. Stefano Ceri (Politecnico di Milano), for all the stimulating discussions and extensive advices during the writing of the work. She likes to acknowledge the advices and criticisms of Serge Abiteboul (INRIA), Jennifer Widom and Jason McHugh (Stanford University), Yannis Papakonstantinou (UCSD), Dan Suciu (AT&T), Letizia Tanca and Sara Comai (Politecnico di Milano) to an early presentation of this survey.

Dongwon Lee wishes to thank Rick Jelliffe (ASCC) for answering questions regarding Schematron and Michael I. Schwartzbach (BRICS) for his helpful comments on DSD during the writing of this paper. Makoto Murata (IBM Tokyo Research Lab.) also greatly helped authors understand RELAX. Dongwon greatly owes his PhD advisor, Prof. Wesley W. Chu (UCLA), for all the fruitful discussions and detailed feedbacks.

References

[1] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. "Views for Semistructured Data". In *Proc. of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J.Wiener, and J. Widom. "The Lorel Query Language for Semistructured Data". *Int'l J. on Digital Libraries (IJDL)*, 1(1):68–88, Apr. 1997.

[3] P. V. Biron and A. Malhotra (Eds). "XML Schema Part 2: Datatypes". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-2/.

[4] A. Bonifati and S. Ceri. "Comparative Analysis of Five XML Query Languages". *ACM SIGMOD Record*, 29(1):68–77, Mar. 2000.

[8] http://www.cobase.cs.ucla.edu/projects/xpress/comp-examples.html

[9] http://www.elet.polimi.it/Users/DEI/Sections/Compeng/Angela.Bonifati/com p-examples.html

[5] R. Bourret, J. Cowan, I. Macherius, and S. St. Laurent (Eds). "Document Definition Markup Language (DDML) Specification, Version 1.0". W3C Note, Jan. 1999. http://www.w3.org/TR/NOTE-ddml.

[6] T. Bray, C. Frankston, and A. Malhotra (Eds). "Document Content Description for XML". Submission to W3C, Jul. 1998. http://www.w3.org/TR/NOTE-dcd.

[7] T. Bray, D. Hollander, and A. Layman. "Namespaces in XML". W3C Recommendation, Jan. 1999. http://www.w3.org/TR/REC-xml-names/.

[8] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). "Extensible Markup Language (XML) 1.0", Feb. 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[9] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). "Extensible Markup Language (XML) 1.0 (2nd Edition)". W3C Recommendation, Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[10] D. Brickley and R. V. Guha (Eds). "Resource Description Framework (RDF) Schema Specification 1.0". W3C Recommendation, Mar. 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.

[11] L. Buck, C. F. Goldfarb, and P. Prescod. "Datatypes for DTDs (DT4DTD)". W3C Note, Jan. 2000. http://www.w3.org/TR/dt4dtd.

[12] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. "XML-GL: a Graphical Language for Querying and Restructuring WWW Data". In *Int'l World Wide Web Conf. (WWW)*, Toronto, Canada, May 1999.

[13] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu (Eds). "XQuery 1.0: An XML Query Language". W3C Working Draft, Jun. 2001. http://www.w3.org/TR/2001/WD-xquery-20010607/.

[14] D. Chamberlin, J. Robie, and D. Florescu. "Quilt: An XML Query Language for Heterogeneous Data Sources". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, May 2000.

[15] J. Clark. "TREX – Tree Regular Expressions for XML". Web page, 2001. http://www.thaiopensource.com/trex/.

[16] J. Clark and S. J. DeRose (Eds). "XML Path Language (XPath) Version 1.0". W3C Recommendation, Apr. 1999. http://www.w3.org/TR/xpath.

[17] J. Clark and M. Murata (Eds). "RELAX NG Tutorial". OASIS Working Draft, Jun. 2001. http://www.oasis-open.org/committees/relax-ng/tutorial.html.

[18] J. Clark (Eds). "XML Transformations (XSLT) Version 1.0". W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt.

[19] J. Clark (Eds). "XML Transformations (XSLT) Version 1.1". W3C Working Draft, Dec. 2000. http://www.w3.org/TR/xslt11.

[20] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. "Your Mediators need Data Conversion! ". In *ACM SIGMOD*, pages 177–188, Seattle, 1998.

[21] S. Comai, E. Damiani, and P. Fraternali. "Graphical Queries over XML Data". In *IEEE Trans. on Information Systems (TOIS)*, (To appear), 2001.

[22] R. L. Costello and J. C. Schneider. "Challenge of XML Schemas - Schema Evolution". Web page, May 2000. http://www.xfront.org/EvolvableSchemas.html.

[23] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, and K. Schwarzhof. "Schema for Object-Oriented XML 2.0". W3C Note, Jul. 1999. http://www.w3.org/TR/NOTE-SOX.

[24] S. J. DeRose, E. Maler, and R. Daniel (Eds). "XML Pointer Language (XPointer) Version 1.0". W3C Last Call Working Draft, Jan. 2001. http://www.w3.org/TR/xptr.

[25] S. J. DeRose, E. Maler, and D. Orchard (Eds). "XML Linking Language (XLink) Version 1.0". W3C Proposed Recommendation, Dec. 2000. http://www.w3.org/TR/xlink/.

[26] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, Dec, 1998. http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[27] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "A Query Language for XML". In *Int'l World Wide Web Conf. (WWW)*, Toronto, Canada, May 1999.

[28] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler (Eds). "XQuery 1.0 Formal Semantics". W3C Working Draft, Jun. 2001. http://www.w3.org/TR/2001/WD-query-semantics-20010607/.

[29] M. Fernandez and J. Robie (Eds). "XQuery 1.0 and XPath 2.0 Data Model". W3C Working Draft, Jun. 2001. http://www.w3.org/TR/2001/WD-query-datamodel-20010607/.

[30] C. Frankston and H. S. Thompson. "XML-Data Reduced". Web page, Jul. 1998. http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm.

[31] W3C XSL Working Group. "The Query Language Position Paper of the XSLT Working Group". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, Dec. 1998.

[32] H. Hosoya and B. C. Pierce. "XDuce: A Typed XML Processing Language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, May 2000.

[33] ISO TC184/SC4/WG11 N101. *"Product data representation and exchange: Implementation methods: XML representation of EXPRESS-driven data"*, Dec. 1999. http://www.eurostep.com/files/wg11n101.pdf.

[34] ISO/IEC. *"Information Technology – Text and Office Systems – Regular Language Description for XML (RELAX) – Part 1: RELAX Core"*, 2000. DTR 22250-1.

[35] R. Jelliffe. "Schematron". Web page, Oct. 2000. http://www.ascc.net/xml/resource/schematron/.

[36] N. Klarlund, A. Moller, and M. I. Schwatzbach. "Document Structure Description 1.0". Specification, 1999. http://www.brics.dk/DSD/dsddoc.html.

[37] N. Klarlund, A. Moller, and M. I. Schwatzbach. "DSD: A Schema Language for XML". In *ACM SIGSOFT Workshop on Formal Methods in Software Practice*, Portland, OR, Aug. 2000.

[38] O. Lassila and R. R. Swick (Eds). "Resource Description Framework (RDF) Model and Syntax Specification". W3C Recommendation, Feb. 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[39] S. St. Laurent. "Describing Your Data: DTDs and XML Schemas". Web page, Dec. 1999. http://www.xml.com/pub/1999/12/dtd/.

[40] A. Layman, E. Jung, E. Maler, H. S. Thompson, J. Paoli, J. Tigue, N. H. Mikula, and S. J. DeRose. "XML-Data". W3C Node, Jan. 1998. http://www.w3.org/TR/1998/NOTE-XML-data.

[41] D. Lee and W. W. Chu. "Comparative Analysis of Six XML Schema Languages". *ACM SIGMOD Record*, 29(3):76–87, Sep. 2000.

[42] B. Ludäescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. "View Definition and DTD Inference for XML". In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, Jan. 1999.

[43] A. Malhotra and M. Maloney (Eds). "XML Schema Requirements". W3C Note, Feb. 1999. http://www.w3.org/TR/NOTE-xml-schema-req.

[44] Microsoft. "XML Schema Developer's Guide". Web page, May 2000. http://msdn.microsoft.com/xml/XMLGuide/schema-overview.asp.

[45] N. Miloslav. "Schematron Tutorial". Web page, May 2000. http://www.zvon.org/HTMLonly/SchematronTutorial/General/contents.html.

[46] S. Muench and M. Scardina (Eds). "XSLT Requirements Version 2.0". W3C Working Draft, Feb. 2001. http://www.w3.org/TR/xslt20req.

[47] M. Murata. "Hedge Automata: a Formal Model for XML Schemata". Web page, 2000. http://www.xml.gr.jp/relax/hedge_nice.html.

[48] M. Murata. "RELAX (REgular LAnguage description for XML)". Web page, Aug. 2000. http://www.xml.gr.jp/relax/.

[49] D. Raggett. "Assertion Grammars". Web page, May 1999. http://www.w3.org/People/Raggett/dtdgen/Docs/.

[50] J. Robie. "The design of XQL". Web page, 1999. http://www.w3.org/Style/XSLT/Group/1998/09/XQL-design.html.

[51] J. Robie. "XQL FAQ". Web page, 1999. http://metalab.unc.edu/xql/.

[52] J. Robie. "XQL Tutorial". Web page, 1999. http://metalab.unc.edu/xql/xql-tutorial.html.

[53] J. Robie, J. Lapp, and D. Schach. *"XML Query Language (XQL)"*. WWW The Query Language Workshop (QL), Cambridge, MA, Dec. 1998. http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[54] J. Robie (Eds). "XQL (XML Query Language)". Web page, Aug. 1999. http://www.ibiblio.org/xql/xql-proposal.html.

[55] A. Sahuguet. "Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, May 2000.

[56] D. Schach, J. Lapp, and J. Robie. "Querying and Transforming XML". In *WWW The Query Language Workshop (QL)*, Cambridge, MA, Dec. 1998.

[57] A. Silberschatz, H. F. Korth, and S. Sudarshan. *"Database System Concepts"*. McGraw-Hill Co., 3rd edition, 1997.

[58] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn (Eds). "XML Schema Part 1: Structures". W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1/.

[59] L. Wood, A. Le Hors, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, G. Nicol, J. Robie, R. Sutor, and C. Wilson (Eds). "Document Object Model (DOM) Level 1 Specification Version 1.0". W3C Recommendation, Oct. 1998. http://www.w3.org/TR/REC-DOM-Level-1/.

Features	DTD	XML-Schema	RELAX	SOX	Schematron	DSD
syntax in XML	No	Yes	Yes	Yes	Yes	Yes
namespace	No	Yes	No	Yes	Yes	No
include	No	Yes	Yes	Yes	No	Yes
import	No	Yes	No	Yes	No	No
built-in type	10	45	45	17	0	0
user-defined type	No	Yes	Yes	Yes	Partial	Yes
type domain constraint	No	Yes	Yes	Partial	Yes	Yes
explicit null	No	Yes	No	No	No	No
simple type by extension	No	No	No	No	No	No
simple type by restriction	No	Yes	No	Yes	No	No
complex type by extension	No	Yes	No	Yes	No	No
complex type by restriction	No	Yes	No	No	No	No
attribute default value	Yes	Yes	Partial	Yes	No	Yes
choice among attributes	No	No	No	No	Yes	Yes
optional vs. required attributes	Yes	Yes	Yes	Yes	Yes	Yes
attribute domain constraint	Partial	Yes	Yes	Partial	Yes	Yes
element default value	No	Partial	No	No	No	Yes
element content model	Yes	Yes	Yes	Partial	Yes	Yes
choice among elements	Yes	Yes	Yes	Yes	Yes	Yes
min & max occurrence	Partial	Yes	Partial	Yes	Yes	Partial
ordered list	Yes	Yes	Yes	Yes	Yes	Yes
unordered list	No	Yes	No	No	Yes	Yes
uniqueness for attribute	Yes	Yes	Yes	Yes	Yes	Yes
uniqueness for non-attribute	No	Yes	No	No	Yes	No
key for attribute	Yes	Yes	Yes	Yes	Yes	Yes
key for non-attribute	No	Yes	No	No	Yes	No
foreign key for attribute	Partial	Yes	Partial	Partial	Yes	Yes
foreign key for non-attribute	No	Yes	No	No	No	Yes
attr-attr	No	No	No	No	Yes	Yes
elm-attr	No	No	Yes	No	Yes	Yes
elm-elm	No	No	Yes	No	Yes	Yes
open model	No	No	No	No	Yes	No
version	No	No	Yes	No	No	Yes
documentation	No	Yes	Yes	Yes	Yes	Yes
embedded HTML	No	Yes	Yes	Yes	Partial	Yes
self-describability	No	Yes	Yes	No	Yes	Yes

Table 1: Comparison summary of the six XML schema languages.

Features	Lorel	XML-QL	XML-GL	XSLT	XQL	XQuery
data model	Graph	Graph	Graph	Tree	Tree	Sequence
differential management of IDREFs	Yes	No	No	No	No	Yes
DM compatibility with related W3C stds	No	No	No	No	No	Yes
document selection	Yes	Yes	Yes	Yes	Yes	Yes
full-fledged joins	Yes	Yes	Yes	Yes	Partial	Yes
dereferencing IDREFs	Yes	No	Yes	Yes	Yes	Yes
multiple output	No	No	No	Yes	No	Yes
definition of views	Yes	No	No	No	No	No
partially specified path expr.	Yes	Yes	Partial	Yes	Yes	Yes
halt on matching cyclic data	Yes	No	Yes	No	No	No
existential quantification	Yes	Yes	Yes	Yes	Yes	Yes
universal quantification	Yes	No	No	No	Yes	Yes
negation	Yes	No	Yes	Yes	Yes	Yes
reduction	No	No	No	Yes	No	No
filtering	No	No	No	Partial	No	Yes
building new XML data	Yes	Yes	Yes	Yes	No	Yes
grouping	Yes	No	Yes	Partial	Yes	Yes
skolem functions	Yes	Yes	Partial	No	No	Yes
aggregates	Yes	No	Yes	Partial	Partial	Yes
nested queries	Yes	Yes	No	Yes	Yes	Yes
set operations	Yes	Partial	Yes	Yes	Yes	Yes
ordering the result	Yes	Yes	Yes	Yes	No	Yes
order-preserving result	Yes	Yes	Yes	Yes	Yes	Yes
querying the order of elements	No	Yes	Yes	Yes	Yes	Yes
querying numbered instances	Yes	No	No	Yes	Yes	Yes
support of datatypes	Partial	No	No	No	No	Yes
type coercion	Yes	No	No	No	Yes	Yes
support of functions	No	Partial	No	Partial	Partial	Partial
support of RDF	No	No	No	No	No	No
support of XPointer & XLink	No	No	No	No	No	No
support of namespaces	No	Partial	No	Yes	Yes	Yes
tag variables	Yes	Yes	No	Yes	No	Yes
update language	Yes	No	Yes	Yes	No	No

Table 2: Comparison summary of the six XML query languages