XML Standards for Ontology Exchange

Last update*: 16 November 2000*

The latest version is available at the OntoText site http://www.ontotext.com/
Comments and suggestions could be sent to marin@sirma.bg

Table Of Contents

1 Introduction

The purpose of this section is to describe the basic ideas standing behind XML, its importance and applications.

1.1 What is XML?

XML stands for “eXtensible Markup Language” - but the acronym is widely used both for the language itself and for a set of related technologies.

The narrow term “XML” stands for the language used to describe data and document structure.

Like most languages XML has a set of rules and conventions that define valid elements of an XML document, and how they could be combined to form a valid document.

On the other hand, XML is a “markup” language i.e. a language that contains special elements (“markups”) used to describe the structure/formatting of parts of the document. Examples of markup languages are HTML, RTF and PostScript. At first glance XML is very similar to HTML , actually XML is a superset of HTML and (most) HTML documents could be considered as XML documents (with some exceptions).

Last but not least, unlike most markup languages, XML is “eXtensible” – the set of elements that could be used in a document is not fixed. Thus you gain the flexibility to create XML documents in the way most suitable for your specific needs. You are not forced to conform to some limited or predefined vocabulary as with HTML for example (of course you have to follow the rules for creating a well-formed and/or valid XML document).

The broad term is used for a set of specifications defining a way to create, transform, render, query and link XML documents. No doubt the core set of XML technologies will emerge and evolve over time as they become more and more adopted. Currently the core set consists of the following standards and proposals:

- **XML** - the specification for the language itself
- **XSchema/DTD** – the way to specify the structure of an XML document, all documents conforming to some structure are said to be “valid”
- **XSLT** – the way to transform XML documents. Very powerful, allows any XML document to be transformed into virtually any type of document (XML, HTML, WML, PDF, RTF, etc). At present it is used mostly for transforming XML into HTML, so that XML content could be published on the Web
- **XLink** - specifies how XML documents (or fragments of documents) are related/linked to each other
- **XPath/XQL/XML-QL** - query languages that are used to perform searches in XML documents and retrieve parts of them according to some criteria

1.2 Why XML?

"This is an era of great intellectual ferment. It is the collision between three cultures: the everything-is-a-document culture, the everything-is-an-object culture, and the everything-is-a-relation culture… XML is one thing all three groups agree on (minus a few details). It, or one of its children, will become the intergalactic dataspeak: the standard way to interchange semistructured and structured data among computer systems"

Jim Gray

Let us outline some of the most important advantages that XML presents:

- open – XML is not designed by a corporation or research group for its specific needs and applications. It is designed by a consortium that aimed at providing the highest possible degree of flexibility and versatility

- extensible – XML has no predefined markup. Actually XML can be considered a meta-language – a language that describes other languages. Of course the new "language" still follows the rules for a well-formed XML document, and all the documents created in the new language are still valid XML documents. Each new language simply defines some vocabulary ("markup") and more specific rules for validity of documents. So we can consider the derived XML languages as a restriction of the general XML syntax, that is most suitable and designed to be used in some specific area. Examples include (but are not limited to)

  Astronomical ML, BiblioML, Bioinformation ML, Chemical ML, Financial Products ML, Gene Expression ML, Geography ML, Horn Logic ML, Human Resources ML, Math ML, Molecular Dynamics ML, Music ML, News ML, Ontology Interchange Language, Ontology ML, Precision Graphics ML, Product Data ML, Relational-Functional ML, Spacecraft ML , User Interface ML, Vector ML, Visa XML, VoiceXML, Wireless ML, XML Corpus Encoding Standard, eXtensible Ontology Exchange Language, eXtensible Scientific Interchange Language, eXtensible User Interface Language

- simple – XML documents are human readable and easy to understand/create. XML removes the complexity of its super language - SGML

- strict syntax – unlike HTML, XML has strict syntax which makes applications dealing with XML documents simpler to implement. Every document that complies with the XML syntax rules is said to be "well-formed"

- separating syntax from semantics – XML defines only the rules for well-formed documents, the semantics that is applied to the document depends solely on the application that processes the document. As there is no predefined set of tags (vocabulary), the same tag (word) may appear in documents conforming to different Document Type Definitions with different meaning. Thus different applications/languages may choose to use a word in the most suitable for their needs manner.

- separates content from presentation – XML does not imply the way of rendering/visualizing the information. In fact (unlike HTML) the content of an XML document might not be intended to be viewed/visualized at all (e.g. two processes/agents exchanging data formatted as XML). This separation gives the possibility of applying different visual presentations to the same XML content (for example the same XML document rendered as HTML for www browsers and as WML for wireless devices)

- designed with internationalization as a goal - XML is designed for multilingual data. Unless specified otherwise, the encoding of an XML document defaults to Unicode, thus giving the ability to store any kind of character in the document

- Information is more usable, because it is described better. Unlike HTML/PostScript that aim to describe the visual layout of a document, XML is more data oriented – it describes data, its properties and relations to other data. For example XML enabled search engines are much more powerful compared to general purpose WWW engines

- Content could be richer, because document linking capabilities of XML are richer than those of HTML

- Content may be distributed across several data stores (servers) - an XML document could include fragments located on remote servers, forming a kind of distributed database.

1.3 Links

1) "Frequently Asked Questions about the Extensible Markup Language" (http://www.ucc.ie/xml/)
2) XML Info – XML resource repository (http://www.xmlinfo.com/)
3) Ken Sall's "XML: Structuring Data for the Web" (http://www.cen.com/ng-html/xml/unix/index.htm)

2 XML technologies

2.1 Core XML

XML is markup language, it consists of text representing the content of the document and markup tags that add some information about the content. An XML document consists of elements, text, comments, unparsed character data, processing instructions and entities.

An *element* consists of start-tag, text and/or nested elements and finally end-tag. Examples might look like:

```
<AUTHOR>William Wharton</AUTHOR>
```

```
<AUTHOR>
        <FIRST_NAME>William</FIRST_NAME>
        <LAST_NAME>Wharton</ LAST_NAME >
</AUTHOR>
```

The name of end-tag is always composed of a "/" concatenated with the name of the start-tag. Tag names are case sensitive (XML documents are case sensitive) and could consist only of letters, digits, dots, underscores or hyphens. Elements that have no content (text or child elements) could leave the end-tag and take the special form <SOMETAG/> which is equivalent to <SOMETAG></SOMETAG>. Every XML document has a *root element* i.e. element that contains all other elements.

When elements are nested (one element contains child elements) one should take care of the proper sequence of the end-tags – elements cannot overlap and one element is always fully contained in another element.

Elements can have *attributes*. Attribute names should comply with the restrictions for tag names. Every attribute has value which is enclosed in apostrophes or double-quotes. For example we could rewrite the above example so that we make use of attributes:

```
<AUTHOR first_name=”William” last_name=”Wharton” />
```

In most cases one has the choice to use child elements or attributes to represent some atomic data (in our examples we have represented "first name" and "last name" both as child elements and as attributes). More complex structures could not be represented by attributes but by child elements.

There are characters that cannot be used within the text of an XML element or within the value of an attribute. Such characters should be “escaped”. The following table gives details:

Character	Escaped sequence
<	<
>	>
&	&
“	"
‘	'

Comments look like <!— *comment content here* --> and they cannot appear inside a markup (within the start or end-tag). Comments cannot contain the string “--”.

The *CDATA sections* make it possible to create texts which are not processed by the XML parser - they may contain tags which will not be recognized as such, ampersands/apostrophes/etc need not be escaped i.e. everything is recognized just as a sequence of characters. Such sections look like <![CDATA[*…content here will not be parsed ….*]]>

Processing instructions allow a document to contain instructions for the application that processes it. The specification does not associate any semantics with processing instructions – they are sent to the processing application by the XML parser. An example for a processing instruction could be <?APPTARGET param1 param2 param3?>. Here APPTARGET is the name of the target application that is expected to handle this processing instruction while the rest is just additional data that is passed with the instruction (it may consist of arbitrary number of parameters).

Entities could be considered as an abbreviation for a part of the document that is commonly used. Internal entities are used as abbreviation for text, they are defined in the DTD of the document (see next chapter). The definition could look like <!ENTITY ak “Atanas Kiryakov”>. Here "ak" is the name of the entity that will be used later in the XML document instead of retyping the text “Atanas Kiryakov” many times. An example could look like:

```
<BOOK>
        <AUTHOR>&ak;</AUTHOR>
        <COMMENT> This is the first book written by &ak;</COMMENT>
</BOOK>
```

Which is equivalent to the following fragment:

```
<BOOK>
        <AUTHOR>Atanas Kiryakov</AUTHOR>
```

```
        <COMMENT> This is the first book written by Atanas Kiryakov </COMMENT>
</BOOK>
```

Note that the entity name is placed within the “&” and “;” symbols when used in an XML document.

Above we described most of the rules that an XML document should follow:

- every start-tag should have a corresponding end-tag (or use the short tag syntax)
- tags are nested properly and they do not overlap
- tag and attribute names contain only valid characters
- every XML document has a root element containing all other elements
- “<”, “>”, “&”, quotation marks and apostrophes are replaced with the appropriate entities
- the value of attribute is enclosed in apostrophes or double quotes

Every document that follows the rules for the XML syntax is said to be *well-formed*

2.1.1 Links

1. Annotated XML Specification (http://www.xml.com/axml/axml.html)
2. XML Specification 1.0 (second edition) (http://www.w3.org/TR/REC-xml)
3. “XML Tutorial” (http://www.zvon.org/xxl/XMLTutorial/General/book_en.html)

2.2 Structure of an XML document - Document Type Definitions

We have already discussed well-formed documents – a document is said to be well-formed if it complies with the XML syntax. Of course we usually need to enforce some rules about the entities that are modeled by the document, i.e. which the valid elements (tags) for a document are and how they could be combined together. The grammatical structure of an XML document is specified by its Document Type Definition (DTD). A document that complies with its DTD is said to be *valid*. A document could be well formed (i.e. follow the XML syntax rules) yet not valid (there might not be DTD specified at all or the document structure may be different from the structure mandated by the DTD).

The DTD language is not valid XML itself (a design flaw that is fixed in XML Schema language). Let us take a look at an example:

```
<? xml version = “1.0” encoding=”UTF8” ?>

<!ELEMENT BOOK          (TITLE, AUTHOR*, ISBN, PUBLISHER , VENDOR_LIST?)>
<!ELEMENT TITLE         (#PCDATA)>
<!ELEMENT ISBN          (#PCDATA)>
<!ELEMENT PUBLISHER     (#PCDATA)>

<!ELEMENT AUTHOR        (FULL_NAME | (FIRST_NAME, LAST_NAME)))>
<!ELEMENT FIRST_NAME    (#PCDATA)>
<!ELEMENT LAST_NAME     (#PCDATA)>

<!ELEMENT VENDOR_LIST   (VENDOR+)>
<!ELEMENT VENDOR        (NAME,PHONE*,BOOK_PRICE)>
<!ATTLIST VENDOR        order           CDATA    #REQUIRED
                        availableOnline (yes|no) “yes” >

<!ELEMENT NAME          (#PCDATA)>
<!ELEMENT PHONE         (#PCDATA)>
```

```
<!ELEMENT BOOK_PRICE    (#PCDATA)>
```

A DTD is composed of element and attribute definitions. Element definitions indicate the name of the element and its type (if it has no child elements) or the child elements and the sequence in which they appear. For example the line `<!ELEMENT BOOK (TITLE, AUTHOR*, ISBN, VENDOR_LIST?)>` indicates that the BOOK element is composed of four child elements appearing in the following sequence : TITLE, AUTHOR (zero or more occurrences) , ISBN, PUBLISHER, VENDOR_LIST (optional).

The number of occurrences of a child element is specified by:

Occurrence symbol	Meaning
?	Zero or one time
*	Zero or more times
+	One or more times
<nothing>	Exactly once

Let us take a look at the AUTHOR element : `<!ELEMENT AUTHOR (FULL_NAME | (FIRST_NAME, LAST_NAME))>`
The "|" symbol means "or" the "," symbol means "and", so the element structure could be read as "the AUTHOR element is composed of either one FIRST_NAME *and* one LAST_NAME child elements *or* by one FULL_NAME child element"

Valid XML fragments for AUTHOR could be:

```
<AUTHOR>
        <FULL_NAME>Atanas Kiryakov</FULL_NAME>
</AUTHOR>
```

or

```
<AUTHOR>
        <FIRST_NAME>Atanas</FIRST _NAME>
        <LAST_NAME> Kiryakov </LAST _NAME>
</AUTHOR>
```

Some elements cannot contain child elements but only text. They are defined as parsed character data #PCDATA. If an element is defined as ANY then it could contain any child elements and/or parsed character data. If the element cannot contain child elements or character data it must be defined as EMPTY.

XML elements can have attributes. Attributes are defined with the <ATTLIST> tag in the DTD. In our sample DTD only the VENDOR element has attributes defined – "order" which is character data and is mandatory and "availableOnline" which accepts only one of the two enumerated values ("yes", "no") and if not specified defaults to "yes"

Attributes may have the following occurrences:

Attribute Type	Meaning
#REQUIRED	Mandatory attribute
#FIXED	The value is fixed
#IMPLIED	Optional attribute

How do XML documents indicate the DTD to which they conform? That is the purpose of the <!DOCTYPE> tag in the beginning of an XML document, for example if we have stored the DTD in a file called book.dtd in the some_directory directory we should have put the following line in the beginning of the document:

```
<!DOCTYPE BOOK SYSTEM "/some_directory/book.dtd">
```

or if we have put the DTD on some internet server we could have used the line :

```
<!DOCTYPE BOOK PUBLIC "-// Sirma AI // book sample dtd //EN" "http://pillango.sirma.bg/book.dtd">
```

The following document is *valid* in the sample DTD we have created:

```
<?xml  version="1.0" encoding="UTF8"?>
<!DOCTYPE BOOK SYSTEM "/xml/book.dtd">

<BOOK>

	<TITTLE>Birdy</TITLE>

	<AUTHOR>
		<FULL_NAME>William Wharton</ FULL_NAME >
	</AUTHOR>

	<ISBN>0679734120</ISBN>

	<PUBLISHER>Knopf</ PUBLISHER >

	<VENDOR_LIST>

		<VENDOR order="1" availableOnline="yes">
			<NAME>Amazon.com</NAME>
			<PHONE> (800)555-1212 </ PHONE >
			<PHONE> (800)555-1313 </ PHONE >
			<BOOK_PRICE>$24.95</ BOOK_PRICE >
		</VENDOR>

		<VENDOR order="2" availableOnline="no">
			<NAME>Border's </NAME>
			<PHONE> (800)615-1313 </ PHONE >
			<BOOK_PRICE >$22.36</ BOOK_PRICE >
		</VENDOR>

	</VENDOR_LIST>

</BOOK>
```

Certainly one may embed the DTD in the XML document itself using the final version of the DOCTYPE tag - <!DOCTYPE BOOK [*copy of the DTD content (without the leading <?xml ...?> tag) goes here*]> but this is not recommended because it makes DTD sharing between documents impossible.

2.2.1 Links

1. Annotated XML Specification (http://www.xml.com/axml/axml.html)
2. "DTD Tutorial" (http://www.zvon.org/xxl/DTDTutorial/General/book.html)
3. Visual DTD – DTD editor from IBM (http://alphaworks.ibm.com/aw.nsf/techmain/visualxmltools)
4. XML Spy XML Editor (http://new.xmlspy.com/)
5. Schema.Net – DTD repository (http://www.schema.net/)
6. XML.org – DTD/Schema repository (http://www.xml.org/registry/)
7. DTD.com – DTD repository (http://www.dtd.com/dtdfactory.shtml)

2.3 XML Namespaces

It is possible that an XML document contains elements (tags) that are intended to be processed from different applications. Imagine that we have the sample document:

```
<?xml version="1.0" encoding="UTF8" ?>
<BOOK>

        <TITTLE>Pride</TITLE>

        <AUTHOR>
                <NAME>William Wharton</NAME>
        </AUTHOR>

        <ISBN>0679734120</ISBN>

        <PUBLISHER>Knopf</ PUBLISHER >

        <VENDOR_LIST>

                <VENDOR order="1">
                        <NAME>Amazon.com</NAME>
                        <PHONE> (800)555-1212 </ PHONE >
                        <PRICE>$12.95</PRICE>
                </VENDOR>

                <VENDOR order="2">
                        <NAME>Border's </NAME>
                        <PHONE> (800)615-1313 </ PHONE >
                        <PRICE>$10.36</PRICE>
                </VENDOR>

        </VENDOR_LIST>

</BOOK>
```

Here we have the tag <NAME> used in two different contexts – it can contain either the name of the author, or the name of some book vendor. If our application needs to apply different meanings to elements (tags) depending on context - for example scan documents and extract the names of all authors available - we will be in trouble. This is because we need to be sure that the <NAME> tag we are processing really contains the name of the author, and not the name of the vendor. Of course in our case we could have replaced NAME with two new elements – AUTHOR_NAME and VENDOR_NAME and make sure our application looks for AUTHOR_NAME. Unfortunately often we do not have control over the vocabulary of our XML documents.

That is why XML Namespaces were designed – to solve the problem of distinguishing elements when used in different context. When using namespaces, every XML element name is prefixed with the namespace it belongs to. So we could define two namespaces: **person** (denoting attributes that are applicable to persons) and **company** (attributes applicable to companies) then we could change the document to:

```
<?xml version="1.0" encoding="UTF8"?>
<BOOK   xmlns:person="persons.dtd"
        xmlns:company="companies.dtd">

        <TITTLE>Pride</TITLE>

        <AUTHOR>
                <person:NAME>William Wharton</NAME>
        </AUTHOR>

        <ISBN>0679734120</ISBN>
```

```
<VENDOR_LIST>

        <VENDOR order="1">
                <company:NAME>Amazon.com</NAME>
                <PHONE> (800)555-1212 </ PHONE >
                <PRICE>$12.95</PRICE>
        </VENDOR>

        <VENDOR order="2">
                <company:NAME>Border's </NAME>
                <PHONE> (800)615-1313 </ PHONE >
                <PRICE>$10.36</PRICE>
        </VENDOR>

</VENDOR_LIST>

</BOOK>
```

In this document we have defined two namespaces - "person" and "company". The definition of a namespace has the form xmlns:xxx="yyy" , where "xxx" will be the namespace prefix that will be used from all elements from this namespace and "yyy" is a unique identifier for this namespace. The specification demands that namespace identifiers be URIs but it is not necessary for the URI to really point to some resource on the net, it is used solely as a unique identifier so that namespaces could be distinguished.

2.3.1 Links

1. XML Namespaces Specification (http://www.w3.org/TR/REC-xml-names)
2. "Namespace Tutorial" (http://www.zvon.org/xxl/NamespaceTutorial/Output/index.html)

2.4 Structure of an XML document – XSchema

The XML Schema language offers another way of specifying the structure of an XML document. It was designed to avoid some of the flaws of the DTD language, for example:

- XML Schemas are defined in XML
- Schemas have more expressive power
- Schemas include native support for namespaces

Like DTDs XML schemas are used as a model for a set of documents. The model describes what combinations of elements, attributes and text are valid and we say that documents conforming to the schema are *valid* in this model. Like DTDs the XML schemas can be considered as a way to define new languages (by defining vocabulary and rules) that applications exchanging information could agree to use.

Unlike DTDs that use a specific language, XML schemas are defined in XML. This makes them easier to use – you don't have to learn another language and you can use existing XML editors to create and modify schemas.

XML Schema language is more powerful than the language for DTD. XML Schema introduces element types, there are about 40 built-in datatypes. They even can be further extended and also one can define complex data types to describe custom structures. Schema types could be inherited and extended (described later). XML schemas allow using regular expressions to specify a pattern that restricts the valid value for a type. The constraints present in the schema language are more powerful - in addition to choice, sequence and iteration (existing in DTD language) one could also use cardinality, length, minimum or maximum value constraints. Moreover the schema language introduces a way to specify a set of unordered child elements which is impossible in DTD.

An XML schema consists mainly of type and element declarations. One would typically first define the types (simple or complex) used in the schema and then define the elements that could compose XML documents (complying with the schema) using the defined types.

There are more than 40 built-in types defined in the XML Schema standard including (but not limited to)

- *boolean*
- *string*
- *float, double, decimal*
- *integer, long, int, short, byte*
- *binary*
- *date, time, month, year, timeDuration*
- *language*
- *uri*
- *id*

Every built-in datatype is characterized by a set of facets, so to create a custom simple type one typically specifies custom values for some of the facets. The set of facets applicable to some type is a subset of {*length, minLength, maxLength, pattern, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, precision, scale, encoding, duration, period*}.

Imagine that we want to create a type representing US phone numbers. The most suitable built-in type is *string*, but if we just use *string*s we cannot guarantee that the phone numbers will consist only of digits and dashes. So we need to create a custom type inherited from *string*. Our new type looks like:

```
<simpleType name=”USPhoneNumber” base=”string”>
        <pattern value=”(1?\d{3}-)? \d{3}-\d{4}” />
</simpleType>
```

This way we can represent numbers such as 1650-444-5555, 650-444-5555 or 444-5555. What we did is simply limit the values valid for the base type by specifying a regular expression for valid US phone numbers – something we could not achieve using DTD.

The general custom type created by inheriting a built-in one will look like:

```
<simpleType name=”MyCustomType” base=”someBasicType”>
        <facet1 value=”value1” />
        <facet2 value=”value2” />
        …
        <facetN value=”valueN” />
</simpleType >
```

where *someBasicType* is a valid built-in type, *facetN* is a valid facet name applicable to the *someBasicType* type and *valueN* is a valid value for the corresponding facet.

Simple types are not always sufficient for describing the structure of an element – an element of a simple type cannot have child elements and attributes. Complex types are used for the task - an example for a complex type is:

```
<complexType name=”PersonType”>
        <sequence>
                <element  name=”TITLE” type=”string” minOccurs=”0” maxOccurs=”1” />
                <element  name=”FIRST_NAME” type=”string” minOccurs=”1” maxOccurs=”1” />
                <element  name=”LAST_NAME” type=”string” minOccurs=”1” maxOccurs=”1” />
```

```
        </sequence>
</complexType>
```

We could define elements from this type now:

```
<element name="PERSON" type=" PersonType" />
```

A valid element of type *PersonType* could be

```
<PERSON>
        <FIRST_NAME>Atanas</FIRST_NAME>
        <LAST_NAME>Kiryakov</LAST_NAME>
</ PERSON >
```

Something similar could be achieved with DTD:

```
<!ELEMENT PERSON  (TITILE?,FIRST_NAME, LAST_NAME)>
```

Now imagine that we need to create a subtype of *PersonType* called *ClarkStudentType* by adding *e-mail* and *nationality* properties and then define elements AUTHOR and COAUTHOR of type *ClarkStudentType*. The only way to create such elements with DTD (you cannot define types in DTD) is to create an element definition like <!**ELEMENT** AUTHOR (TITLE?,FIRST_NAME, LAST_NAME, EMAIL*, NATIONALITY)> copying most of the definition of PERSON (which is error prone) while with XML Schema we could use subclassing:

```
<complexType name="ClarkStudentType" base=" PersonType" derivedBy="extension">
        <sequence>
                <element name="EMAIL" type="string" minOccurs="0" maxOccurs="unbounded" />
                <element name="NATIONALITY" type="string" minOccurs="1" maxOccurs="1" />
        </sequence>
</complexType>
```

We could define elements from this type now:

```
<element name="AUTHOR" type=" ClarkStudentType" />
<element name="COAUTHOR" type=" ClarkStudentType" />
```

Valid XML fragments that represent AUTHOR and COAUTHOR elements of type *ClarkStudentType* could be:

```
<AUTHOR>
        <TITLE>Mr</TITLE>
        <FIRST_NAME>Kiril</FIRST_NAME>
        <LAST_NAME>Simov</LAST_NAME>
        <EMAIL>kivs@bgcict.acad.bg</EMAIL>
        <NATIONALITY>bulgarian</ NATIONALITY>
</ AUTHOR >

<COAUTHOR>
        <TITLE>Mr</TITLE>
        <FIRST_NAME>Atanas</FIRST_NAME>
        <LAST_NAME>Kiryakov</LAST_NAME>
        <EMAIL>naso@sirma.bg</EMAIL>
        <EMAIL>naso@ontotext.com</EMAIL>
        <NATIONALITY>bulgarian</ NATIONALITY>
</ COAUTHOR >
```

Note that the elements of the derived types are always appended to the elements of the base type.

In this example we used inheritance by extension (note the *derivedBy* attribute of the *complexType* element). This means that the subtype defines new elements in addition to those of the parent type. There is another type of inheritance – by restriction, which restricts in some way the elements of the parent type. We could derive a new type – BulgarianClarkStudentType as:

```
<complexType name="BulgarianClarkStudentType" base="ClarkStudentType" derivedBy="restriction">
	<sequence>
		<element name="EMAIL" type="string" minOccurs="0" maxOccurs="unbounded" />
		<element name="NATIONALITY" type="string" fixed="bulgarian" minOccurs="1" maxOccurs="1" />
	</sequence>
</complexType>
```

Pay attention to the *derivedBy* and *fixed* attributes. In this example we restricted the base type by fixing the valid value for NATIONALITY to "bulgarian".

In the preceding examples we extensively used the *sequence* tag. Its purpose is to define a group of elements that always appear together and in the specified order. This symbol is equivalent to the comma symbol in DTD. With DTD one could define a choice of elements by using the "|" symbol, the same effect could be achieved in XML Schema using the <choice> tag.

DTD notation

```
<!ELEMENT PERSON (FULL_NAME | (FIRST_NAME, LAST_NAME)) >
```

XML Schema notation

```
<element name="PERSON">
	<complexType>
		<choice>
			<element name="FULL_NAME" type="string" minOccurs="1" maxOccurs="1" />
			<sequence>
				<element name="FIRST_NAME" type="string" minOccurs="1" maxOccurs="1" />
				<element name="LAST_NAME" type="string" minOccurs="1" maxOccurs="1" />
			</sequence>
		</choice>
	</complexType>
</element>
```

In addition to the *sequence* and *choice* element groupings (also available in DTD) XML schemas allow for describing elements that are grouped together without respect to any particular order (set of elements). This is achieved with the <any> tag that is used in the same way as <choice> and <sequence> tags. One restriction of the elements grouped in a set is that the *maxOccurence* attribute value is always equal to "1".

In the preceding example we demonstrated another feature of XML Schema – *anonymous types* (PERSON is an element of anonymous type), i.e. types that have no names and whose definition is embedded in the definition of an element. This is a convenient way to specify the structure of an element without creating a distinct type. As anonymous types have no names we cannot define several elements of the same anonymous type – which means that they are not reusable.

XML schema supports element attributes just like DTD, of course the attributes can be of any simple type (built-in or custom). Let us extend the ClarkStudentType example by adding mandatory boolean attribute *isLecturer*

DTD notation

```
<!ELEMENT clarkStudent (title?,first_name,last_name,email*,nationality)>
<!ATTLIST   clarkStudent isLecturer (true|false) #REQUIRED>
```

XML Schema notation

```
<complexType name="ClarkStudentType" base=" PersonType" derivedBy="extension">
	<sequence>
		<element name="EMAIL" type="string" minOccurs="0" maxOccurs="unbounded" />
		<element name="NATIONALITY" type="string" minOccurs="1" maxOccurs="1" />
```

```
        </sequence>
        <attribute  name="isLecturer" type="boolean" use="required" />
</complexType>
```

Attribute declarations should always follow element declarations. Attributes could also be grouped together in *attributeGroup*s so that one may avoid repeating the declaration of common attributes in several types and just include a reference to the group in the type definition.

2.4.1 Links

1. XML Schema Tutorial (http://www.xfront.com/xml-schema.html)
2. "Derivation, tolerance and validity – a formal model of type definition in XML Schemas" (http://www.gca.org/papers/xmleurope2000/papers/s06-02.html)
3. XML Schema Part 0: Primer (http://www.w3.org/TR/xmlschema-0/)
4. XML Schema Part 2: Datatypes (http://www.w3.org/TR/xmlschema-2/)
5. XML Schema: An Intensive One-Day Tutorial (http://www.oasis-open.org/cover/thompsonSchemaSlides19991220_files/frame.htm)
6. "The basics of using XML Schema to define elements" (http://www-4.ibm.com/software/developer/library/xml-schema/index.html?l=xmlst,t=gr,p=schema)
7. Oracle XML Schema processor (http://technet.oracle.com/tech/xml/schema_java/index.htm)
8. Apache XML Parser (supports XML Schema) (http://xml.apache.org/xerces-j/)
9. Visual DTD – schema editor from IBM (http://alphaworks.ibm.com/aw.nsf/techmain/visualxmltools)
10. XML Spy XML Editor with Schema support (http://new.xmlspy.com/)
11. Schema.Net – DTD/Schema repository (http://www.schema.net/)
12. XML.org – DTD/Schema repository (http://www.xml.org/registry/)
13. DTD.com – Schema repository (http://www.dtd.com/schemafactory.shtml)
14. "XML Schemas: Best Practices" (http://www.xfront.org/BestPractices.html)

2.5 Working with XML documents – DOM and SAX

XML is a very powerful language but it is useless without a way to process XML documents. The most popular solutions of this problem at present are SAX and DOM, which specify APIs for processing parts or whole XML documents. Free and commercial SAX and DOM implementations are available in every popular programming language. Every XML application that has to extract content from a document or to transform it incorporates some XML parser that:

1. checks syntax of the XML document (the document has to be well-formed, i.e. follow the XML syntax rules)
2. optionally checks validity of the XML document (i.e. whether the XML document conforms to some DTD/schema)
3. finally transforms the raw XML document into some structure understood by the application (DOM) or generate appropriate events for the processing application (SAX)

2.5.1 SAX – Simple API for XML

Neither SAX 1.0 nor SAX 2.0 are World Wide Web Consortium (W3C) standards but these APIs are some of the most popular. SAX is based on a common programming pattern called *callback*, i.e. you are expected to specify some handler which will be invoked to process some kind of event, whenever it occurs. In the case of an XML document events could be the occurrence of an XML tag or block of text or errors.

Error notifications are supplied to the error handler (in terms of Java this is a class that implements the *org.xml.sax.ErrorHandler* interface). Notifications about the work of the parser are sent to the document handler - *org.xml.sax.DocumentHandler* . The types of events that could be received are :

- Start of document (parsing begins)
- End of XML document reached (parsing finished)
- Start of XML element (i.e. <SOMETAG> encountered). The handler is supplied with the name of the tag and the list of the attributes comprising the tag
- End of XML element (i.e. </SOMETAG> encountered). The handler is supplied with the name of the tag.
- Text encountered (either the text between the start-tag and end-tag or CDATA section)
- Processing instruction encountered (e.g. <?pi “do some action”?>)

To summarize , the minimum work an XML application would do is :

1. define the handler for XML events
2. define the handler for errors (optional)
3. register handlers with the SAX parser
4. invoke the SAX parser for some XML file

While simple and easy to use SAX has some shortcomings:

- as the SAX model is event based, the parser will not create any structure representing the XML document, so the application should take care of creating a useful internal representation of the information in the document.
- It could be used only for parsing XML documents, but not for modifying existing documents or creating new ones.

2.5.2 DOM – Document Object Model

DOM is a set of interfaces for accessing tree-structured documents, which makes it very appropriate for processing XML – the DOM parser will create a tree structure representing the XML document being parsed (unlike the SAX parser which will just generate appropriate events). Therefore the main benefit is that the DOM parser will do a little more work than the SAX parser as it will generate some useful representation of the XML data, that could be used by your application.

Processing XML documents with DOM usually includes the following steps :

1. The DOM parser processes the XML document and returns a *Document*, which is a tree of *Node*s representing the XML document. Every node in the tree represents some part of the document such as an element, attribute or the text content of an element (i.e. the text between start-tag and end-tag).

2. The application accesses the root element of the tree, from which it could recursively process the DOM tree (accessing its child elements).

3. The application creates/removes elements, adds/changes attributes, modifies the document structure with the help of the *Document, Node* and *Element* interfaces.

The main interfaces that an application will use while dealing with DOM are:

- **Node** – this is the base type, representing a node in the DOM tree. Gives functionality for accessing child nodes and adding new child nodes.

- **Document** - represents the entire XML document as a tree of *Node*s (the DOM parser will return Document as a result of parsing the XML). Provides direct access to the root element and methods for creating elements, attributes, text, comments, processing instructions (i.e. everything that an XML document can contain)

- **Element –** represents elements of the XML document. Every pair of tags <SOMETAG>…</SOMETAG> forms an *Element* in the DOM tree, and all XML tags between the start-tag and the end-tag will be represented as child elements. The *Element* interface allows querying its attributes as well as modifying/adding/removing attributes

- **Attr** – represents an attribute of some XML element, the interface enables setting/getting the value of that attribute

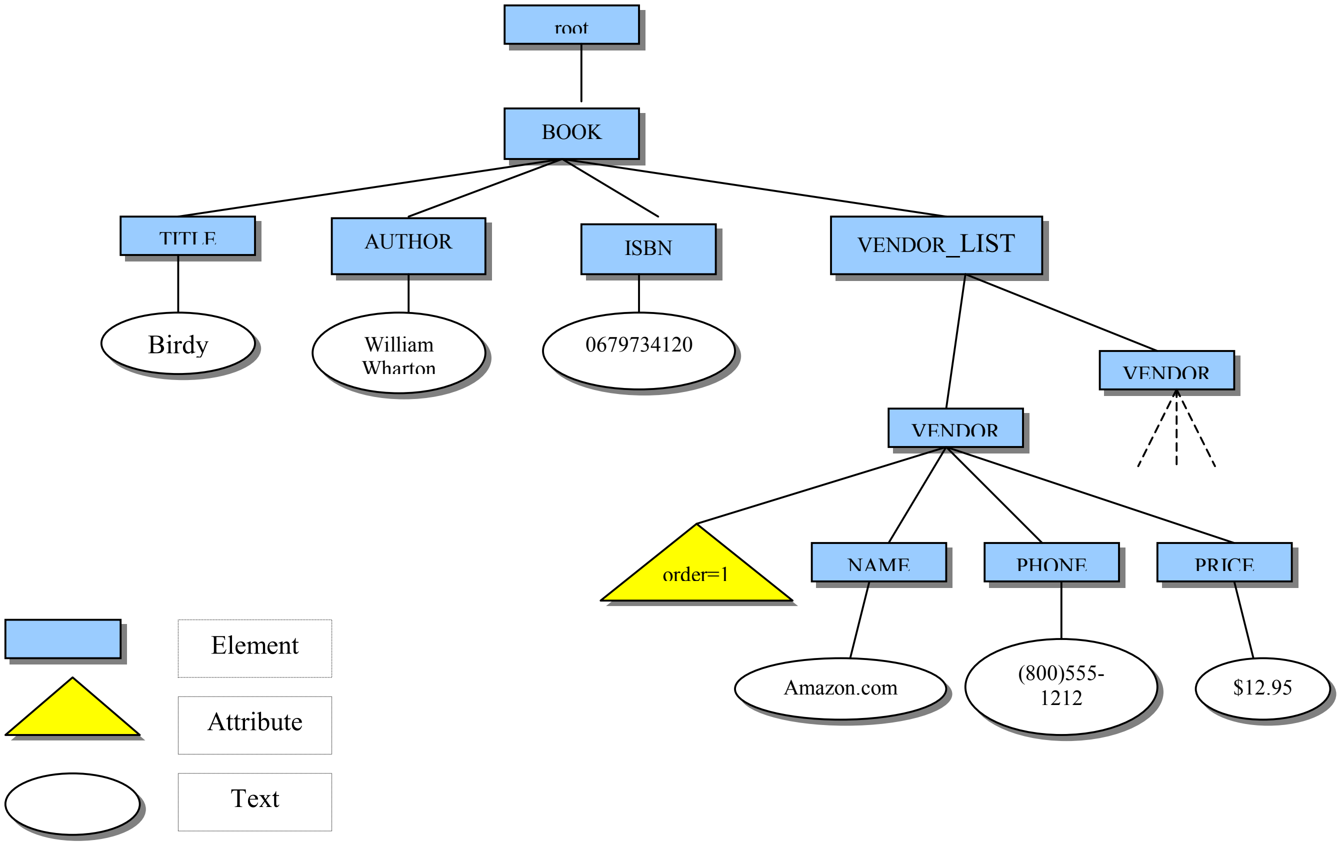
- **Text** – used to represent the text content of an element (i.e. the text between tags, that is not part of any child element). The methods of the interface allow getting/setting/modifying the content.

Details about other DOM interfaces could be found in the specification.

If our application is supposed to process the following XML document:

```
<?xml  version="1.0" encoding="UTF8"?>
<BOOK>
        <TITTLE>Birdy</TITLE>

        <AUTHOR>William Wharton</AUTHOR>

        <ISBN>0679734120</ISBN>

        <VENDOR_LIST>

                <VENDOR order="1">
                        <NAME>Amazon.com</NAME>
                        <PHONE> (800)555-1212 </ PHONE >
                        <PRICE>$12.95</PRICE>
                </VENDOR>

                <VENDOR order="2">
                        <NAME>Border's </NAME>
                        <PHONE> (800)615-1313 </ PHONE >
                        <PRICE>$10.36</PRICE>
                </VENDOR>

        </VENDOR_LIST>

</BOOK>
```

then BOOK will be the root element of the document, TITLE, AUTHOR, ISBN, VENDOR_LIST will be its child elements, TITLE has one child node of type *text* with content "Pride", VENDOR elements have attribute "order".



2.5.3 SAX or DOM?

Depending on the type of XML processing your application will do, either SAX or DOM could be the most appropriate API to use. Actually there is a new emerging API now called JDOM, which is based on DOM but is much easier to use because it is designed for Java (i.e. it is not programming language neutral).

SAX is easier to learn, it is faster, more efficient and has small memory demands (because it is event oriented and does not create any data representation). It is more suitable for large documents that are not changed and need not be represented in the memory as a tree structure. For example if your application is interested only in specific tags from the source XML (filtering, searching) it will be inefficient to create some tree like structure for the document. Because of its event oriented nature SAX cannot be used to access random parts of the document or for backward traversing – the elements are processed in the sequence they occur in the source XML. Also your application cannot create/modify documents using SAX.

DOM is more suitable for applications that will create/modify documents. Also if your application needs some tree-like representation of the source XML document, DOM is the right choice, but bear in mind that it creates a structure representing the whole document. So if the source XML is very large, the memory used by your application will grow proportionally. With the DOM tree your application will have random access to parts of the document.

2.5.4 Links

1. DOM Level 1 Specification (http://www.w3.org/TR/REC-DOM-Level-1/)

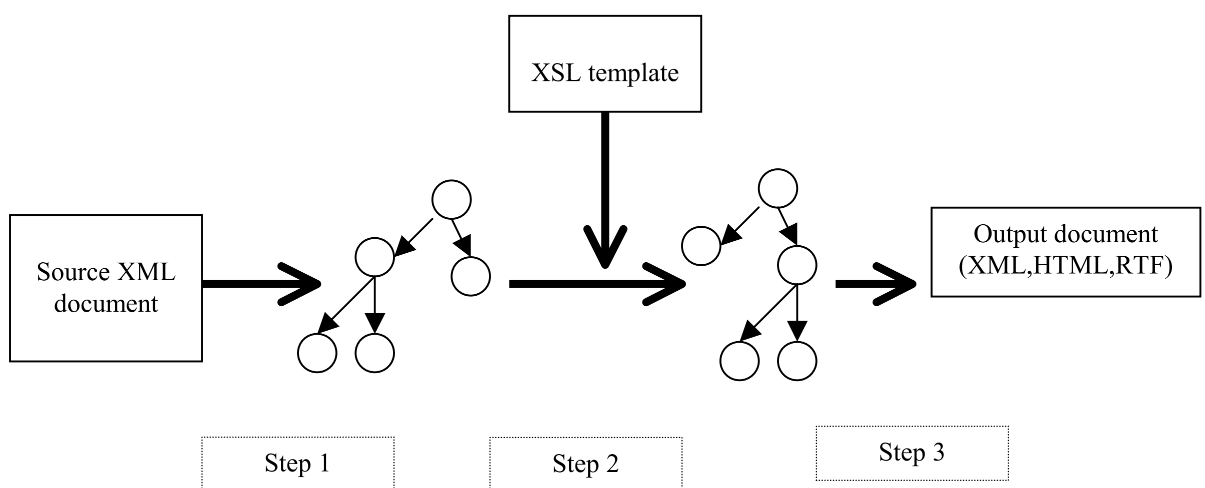
2. DOM Level 2 Specification (http://www.w3.org/TR/DOM-Level-2/)
3. SAX 2.0 specification (http://www.megginson.com/SAX/)
4. "Using the IBM XML Parser (xml4j)" (ftp://www6.software.ibm.com/software/developer/library/xml4j.pdf)
5. "Parsing XML Using Java" (http://www-4.ibm.com/software/developer/education/tutorial-prog/parsing.html)
6. "Mapping XML to Java" (http://www.javaworld.com/javaworld/jw-08-2000/jw-0804-sax_p.html)
7. Oracle XML parser (http://technet.oracle.com/tech/xml/parser_java2/)
8. Apache XML parser (http://xml.apache.org/xerces-j/)
9. XP by James Clark (SAX parser) (http://www.jclark.com/xml/expat.html)
10. JDOM information (http://www.jdom.org/)
11. "Easy Java/XML integration with JDOM" (http://www.javaworld.com/javaworld/jw-07-2000/jw-0728-jdom2_p.html)

2.6 XML Transformations

eXtensible Stylesheet Language for Transformations (XSLT) is a very powerful transformation language. It allows any XML document to be transformed into virtually any type of document (XML, HTML, WML, PDF, RTF, etc). At present it is used mostly for transforming XML into HTML, so that XML content could be published on the Web, but the potential power of XSL is not adding some presentation to XML documents (with the help of HTML or WML) nut in the transformation between XML documents conforming to different vocabularies (DTDs, schemas).

For example with the help of XSLT one ontology represented in some XML based language for ontology exchange could be translated into another language (provided that the languages have similar expressive power) and used by the tools understanding the second language. Actually this is the approach chosen by OIL authors - there are tools (XSL templates) that transform OIL into SHIQ (so that it could be used with the FaCT reasoner) and into RDFS (so that it could be used with any RDFS aware tool)

The process of XSL transformation looks like:



The process includes the following steps: the source document (XML) is translated into some tree representation (step 1), then according to the rules in the XSLT stylesheet the original tree is transformed into a new tree (step 2) and finally the result is serialized to file (XML. HTML, WML, etc.)

This section is incomplete.

2.6.1 Links

1. The XSLT specification (http://www.w3.org/TR/xslt)
2. "XSL Transformations" – chapter 14 from "The XML Bible" (http://metalab.unc.edu/xml/books/bible/updates/14.html)
3. XSLT.com – xslt tools, tutorials and resources (www.xslt.com)
4. "XSL Tutorial" (http://www.xfront.com/xsl.html)
5. "XSLT tutorial" (http://www.brics.dk/~amoeller/XML/xslt.html)
6. "XSLT Reference" (http://www.zvon.org/xxl/XSLTreference/Output/index.html)
7. "Tutorial: Transforming XML Documents" (http://www-4.ibm.com/software/developer/education/transforming-xml/?open&loc=dw-trans,t=int-xml)
8. IBM XSL Editor (http://alphaworks.ibm.com/aw.nsf/techmain/xsleditor)
9. Oracle XSL processor (http://technet.oracle.com/tech/xml/parser_java2/)
10. XT by James Clark (http://www.jclark.com/xml/xt.html)
11. Apache XSL processor (http://xml.apache.org/xalan/)
12. XSLT Test Tool (http://www.netcrucible.com/xslt/xslt-tool.htm)
13. "XML document processing in Java using XPath and XSLT" (http://www.javaworld.com/jw-09-2000/jw-0908-xpath.html)

2.7 Linking XML documents – XLink

The XLink language is used to describe links between XML documents or document fragments. In addition to the unidirectional links available in HTML, XLink adds constructs for links with multiple targets, bi-directional links and links that can be stored out of any of the documents they link together.

This section is incomplete.

2.7.1 Links

1. XLink Specification (http://www.w3.org/TR/xlink/)
2. XLink engine by Step UK (http://www.stepuk.com/x2xdocs/x2xdocs.html)

2.8 Querying XML Documents – XPath , XQL, XML-QL, Quilt

Query languages make it possible to retrieve parts of XML documents, according to some specified restriction/condition. Because of the different approaches chosen, there is a variety of query languages available. At present only XPath is W3C recommendation (see [1]) but there are many powerful tools available for the most popular XML query languages.

XPath takes its name from the way query expressions use to locate fragments in the document - they *path expressions*. If you consider the XML document as a tree structure (remember the DOM model) then a path is the concatenation of the names of the elements from the root element to the chosen child element.

Consider the sample XML document introduced in 2.2:

```
<?xml  version=”1.0” encoding=”UTF8”?>
<!DOCTYPE BOOK SYSTEM “/xml/book.dtd”>

<BOOK>

        <TITTLE>Birdy</TITLE>

        <AUTHOR>
                <NAME>William Wharton</ NAME >
        </AUTHOR>

        <ISBN>0679734120</ISBN>

        <PUBLISHER>Knopf</ PUBLISHER >

        <VENDOR_LIST>

                <VENDOR order=”1” availableOnline=”yes”>
                        <NAME>Amazon.com</NAME>
                        <PHONE> (800)555-1212 </ PHONE >
                        <PHONE> (800)555-1313 </ PHONE >
                        <BOOK_PRICE>$24.95</ BOOK_PRICE >
                </VENDOR>

                <VENDOR order=”2” availableOnline=”no”>
                        <NAME>Border’s </NAME>
                        <PHONE> (800)615-1313 </ PHONE >
                        <BOOK_PRICE >$22.36</ BOOK_PRICE >
                </VENDOR>

        </VENDOR_LIST>

</BOOK>
```

The path expression "/VENDOR_LIST/VENDOR[1]/BOOK_PRICE" selects the <BOOK_PRICE>$24.95</ BOOK_PRICE > element , here the leading "/" locates the root element, then "VENDOR_LIST" locates the <VENDOR_LIST > element, then "VENDOR[1]" locates the first <VENDOR> element and finally "BOOK_PRICE" locates the <BOOK_PRICE > child element of the already located in the previous step <VENDOR> element.

Another XPath expression could be "/*/NAME" which will return a list of the following elements <NAME>William Wharton</ NAME >, <NAME>Amazon.com</NAME> and <NAME>Border’s </NAME>. The "*" symbols stands for any element or sub-path.

An example using attributes could look like "/VENDOR_LIST/VENDOR[@availableOnline="no"]/PHONE" which will return the <PHONE> (800)615-1313 </ PHONE > element (first we select the vendor list, then the vendor with attribute "availableOnline" equal to "no", which in our case is Border's and finally we select the vendor's phone).

XQL is very similar to XPath - it uses path notation too but some of the XPath features are missing, while there are new features not found in XPath.

XML-QL is very different from XPath, it is closer to SQL and it has great expressive power in its ability to not only return the selected elements from the document but to create elements with completely new structure and return them as resulting recordset (like in SQL)

Quilt combines the best ideas from XPath, XML-QL, XQL, SQL and OQL.

This section is incomplete

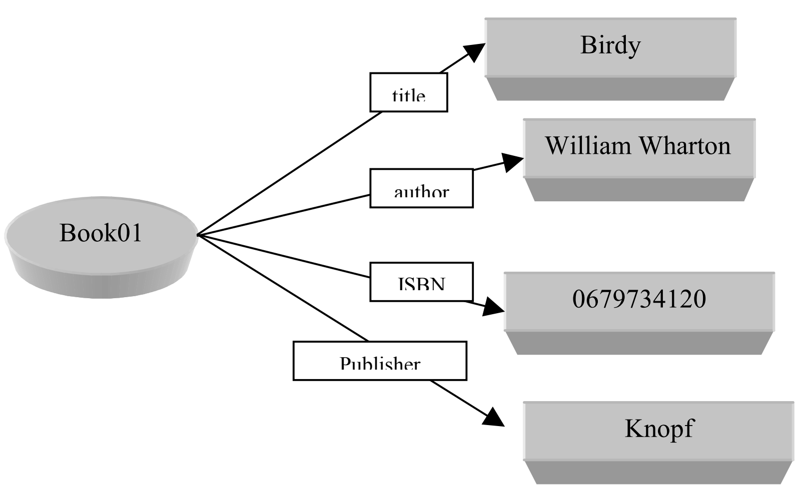
2.8.1 Links

1. XPath Specification (http://www.w3.org/TR/xpath)
2. XQL Proposal (http://www.w3.org/TandS/QL/QL98/pp/xql.html)
3. XQL FAQ (http://metalab.unc.edu/xql/)
4. XQL Tutorial (http://metalab.unc.edu/xql/xql-tutorial.html)
5. XML-QL Proposal (http://www.w3.org/TR/NOTE-xml-ql/)
6. "XML document processing in Java using XPath and XSLT" (http://www.javaworld.com/jw-09-2000/jw-0908-xpath.html)
7. Quilt papers (http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html)
8. "XPath: XML Path Language" (http://www.arbortext.com/Think_Tank/Norm_s_Column/Issue_One/Issue_One.html)

2.9 Resource Description Framework - RDF

The purpose of the Resource Description Framework is to enable encoding and exchange of structured metadata. RDF is the result of an early project to allow web search engines perform not only content searches but also meta searches that best describe and rate the web content. Every RDF document could be considered as a group of statements that describe resources. What is a resource? This is anything that could be identified by a valid URI (Unified Resource Identifier), it may be a web page, web page fragment, XML file, a person, a book etc. Resources are described by their properties, every property has type and value. Property values could be atomic (numbers, strings) or other resources.

Let us take a look at the example XML describing a book from the previous chapters. If we ignore the vendors for a while, we can agree that the statement described by the XML file is "The book named 'Birdy' is written by W.Wharton, published by Knopf, and has ISBN equal to 0679734120 " In other words we have some statement about some book (a resource) which is described by its attributes: title ("Birdy"), author (W.Wharton), ISBN (0679734120), publisher (Knopf).

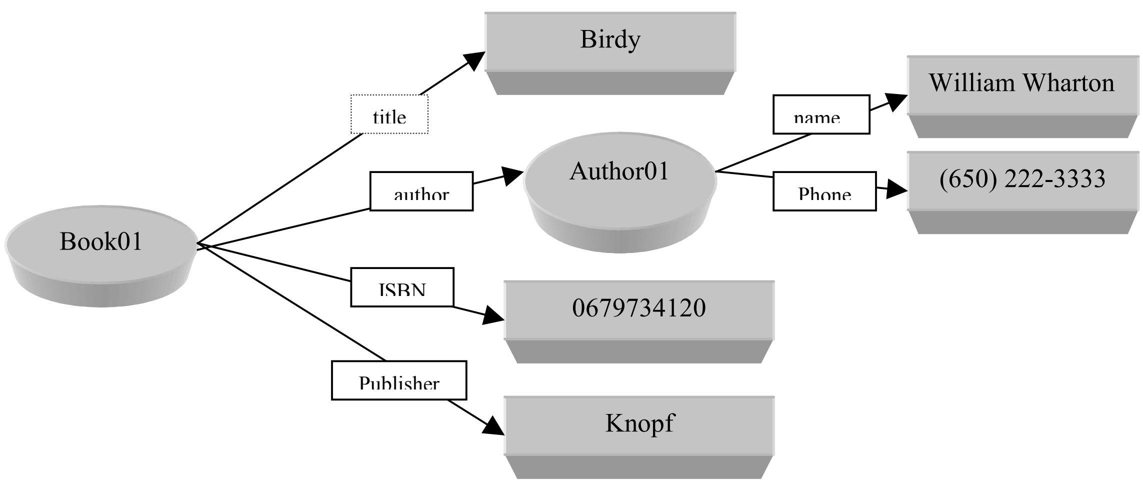


This could be expressed in RDF as:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
        <rdf:Description about="urn:isbn: 0679734120">

                <TITILE>Birdy</TITLE>
                <AUTHOR>William Wharton</AUTHOR>
                <ISBN>0679734120</ISBN>
                <PUBLISHER>Knopf</PUBLISHER>

        </rdf:Description>
</rdf:RDF>
```

Here we have a single statement (description of resource) about a book, identified by the URI "urn:isbn:0679734120" (we could have used any other valid URN for identifying the book). TITLE, AUTHOR, ISBN and PUBLISHER are the names of the elements we choose to represent the attributes of the book (of course we could choose to conform to already established vocabulary like Dublin Core in which case we will have to replace AUTHOR with CREATOR and ISBN with IDENTIFIER).

In the above example we used only atomic attributes. Let us extend the example by turning the book author into resource and adding attributes to it (name and phone).

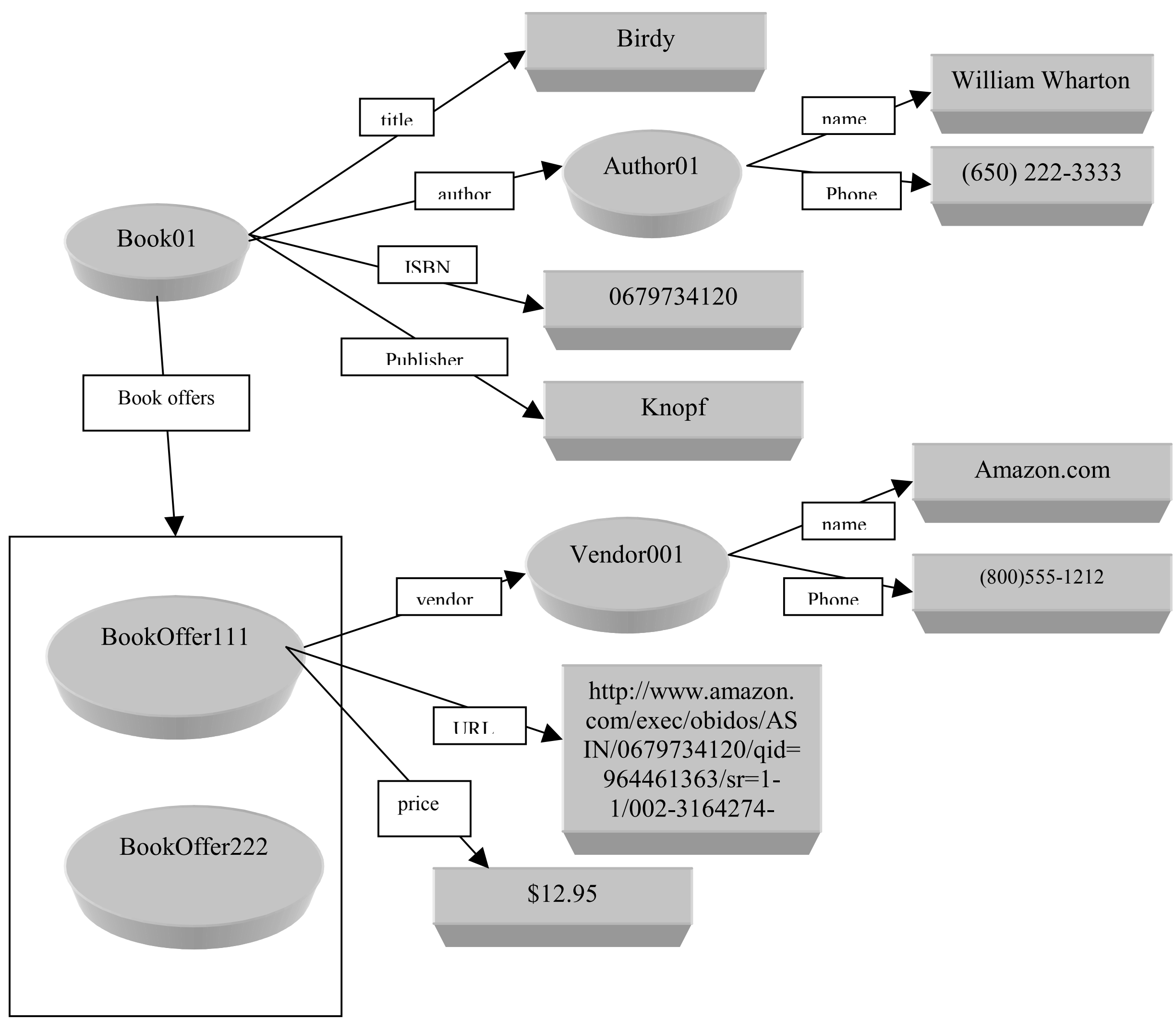


The RDF file will contain two statements about the resources book (identified by "urn:isbn: 0679734120") and author (identified by "mailto:WilliamWharton@hotmail.com"). Notice how the attribute AUTHOR of the book now is not an atomic one but a reference to a resource.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

        <rdf:Description about="urn:isbn: 0679734120">
                <TITILE>Birdy</TITLE>
                <AUTHOR rdf:resource="mailto:WilliamWharton@hotmail.com" />
                <ISBN>0679734120</ISBN>
                <PUBLISHER>Knopf</PUBLISHER>
        </rdf:Description>

        <rdf:Description about="mailto:WilliamWharton@hotmail.com">
                <NAME>William Wharton</NAME>
                <PHONE>(650) 222-3333</ PHONE >
        </rdf:Description>

</rdf:RDF>
```

Finally we could describe book vendors as resources too. We will first introduce RDF containers. Containers are used to describe collection of resources – for example when a resource has multiple properties of the same type ("book vendor" in our case). There are three types of containers – a bag (unordered collection of values, duplicates allowed), a sequence (ordered bag) and a choice (list of alternatives from which one is chosen)

We could now describe vendors as resources with attributes : name, phone and we will put them in a container. We will also introduce a resource for representing the specific book offer from a vendor (described by attributes: price and www page). We end up with something like:



The full RDF file representing these resources and the relations between them looks like:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <rdf:Description about="urn:isbn: 0679734120">
        <TITLE>Birdy</TITLE>
        <AUTHOR rdf:resource="mailto:WilliamWharton@hotmail.com" />
        <ISBN>0679734120</ISBN>
        <PUBLISHER>Knopf</PUBLISHER>

        <BOOK_OFFERS>
            <rdf:Bag>
                <rdf:li resource="#bookOffer111" />
                <rdf:li resource="#bookOffer222" />
            </rdf:Bag>
        </ BOOK_OFFERS >

    </rdf:Description>

    <rdf:Description about="mailto:WilliamWharton@hotmail.com">
        <NAME>William Wharton</NAME>
        <PHONE>(650) 222-3333</ PHONE >
    </rdf:Description>

    <rdf:Description ID="bookOffer111">
        <VENDOR rdf:resource="http://www.amazon.com" />
        <URL> http://www.amazon.com/exec/obidos/ASIN/0679734120/qid=964461363/…</ URL>
        <PRICE>$12.95</PRICE>
    </rdf:Description>

    <rdf:Description ID="bookOffer222">
        <VENDOR rdf:resource="http://www.borders.com" />
        <URL> http://search.borders.com/fcgi-bin/db2www/search/search.d2w/Details…</ URL>
        <PRICE>$10.95</PRICE>
    </rdf:Description>

    <rdf:Description about="http://www.amazon.com">
        <NAME>Amazon.com</NAME>
        <PHONE> (800)555-1212 </ PHONE >
    </rdf:Description>

    <rdf:Description about="http://www.borders.com">
        <NAME>Border's </NAME>
        <PHONE> (800)615-1313 </ PHONE >
    </rdf:Description>

</rdf:RDF>
```

2.9.1 Links

1. "RDF Specification" (http://www.w3.org/TR/rdf-schema/)
2. "RDF Tutorial" (http://www710.univ-lyon1.fr/~champin/rdf-tutorial/)
3. "RDF Tutorial" (http://www.zvon.org/xxl/RDFTutorial/General/book.html)
4. "RDF: A Frame System for the Web" (http://www.ltg.ed.ac.uk/~ht/ora-rdf-dagstuhl.html)
5. Online SWI-Prolog RDF parser (http://swi.psy.uva.nl/projects/SWI-Prolog/packages/sgml/online.html)
6. RDF API by Stanford University (http://www-db.stanford.edu/~melnik/rdf/api.html)
7. Protégé 2000 – KBS Editor supporting RDF (http://smi-web.stanford.edu/projects/protege/index.html)
8. RDF DB – RDF database (http://www.guha.com/rdfdb/)
9. "Storing RDF in a relational database" (http://www-db.stanford.edu/~melnik/rdf/db.html)
10. "Using International Standard Book Numbers as Uniform Resource Names" (http://www.ietf.org/internet-drafts/draft-hakala-isbn-00.txt)

3 XML for ontology sharing

3.1 *Ontologies*

We will stick to the definition of an ontology given in [1] - a shared understanding of some subject area which helps people or processes achieve better communication, inter-operability, effective reuse. The ontology embodies a conceptualization – definitions of entities, their attributes and relationships that exist in some domain of interest. The conceptualization is explicitly represented.

3.1.1 Links

1. Mike Uschold & Michael Gruninger. “Ontologies : Principles, Methods and Applications”. Knowledge Engineering Review , also available from http://www.aiai.ed.ac.uk/~entprise/enterprise/ontology.html
2. D. Fensel: Relating Ontology Languages and Web Standards (http://www.cs.vu.nl/~dieter/ftp/paper/mod2000.pdf)

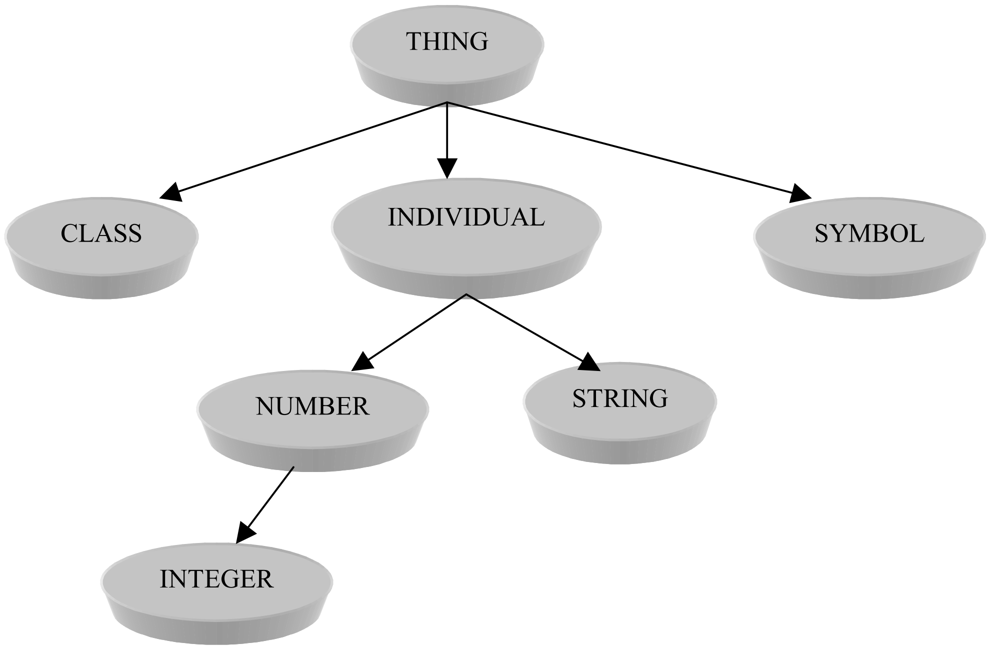
3.2 *XOL – XML-based Ontology exchange Language*

XOL is XML based language for exchange of ontology definitions. The authors have chosen XML because it is simple yet powerful and because of its emerging popularity. The semantics of XOL is based on OKBC. A version of the XOL DTD could be found in Appendix C (the one from the original paper contains some minor errors). We have also created an XML Schema for XOL.

The OKBC knowledge model works with classes, slots, facets and individuals. The atomic datatypes defined in OKBC are:

- integer
- floating point number
- boolean
- string
- name of class (?)

Classes are collections of entities. If the entities of a class are classes themselves, then the class is called *metaclass*. If the entities are not classes, then they are called *individuals*.
The predefined classes in OKBC are THING, CLASS, INDIVIDUAL, SYMBOL, NUMBER, STRING, INTEGER. The THING class is the root of the hierarchy:



Entity membership to some class is presented by the relation *instance-of*, i.e. instance-of(Ind,Cls) is true if Ind is entity of class Cls. There is also an inverse relation – *type-of*. Another relation of interest is *subclass-of* which defines the class hierarchy. The relation is defined only for classes, and *subclass-of*(Csub,Csuper) is true if and only if all instances of Csub are instances of Csuper too. The inverse relation is *superclass-of*.

Every instance of a class has a set of *own slots* associated with it. Every own slot has associated with it a set of *slot values*. We could consider the slot as a binary relation holding between the instance and the value of the own slot. An own slot has associated with it a set of *own facets* which in turn have associated with them sets of *facet values*. For example we could consider the book "Birdy" as an individual of the class BOOK, having slots like AUTHOR, ISBN, PUBLISHER. The slot value of ISBN would be 0679734120 and we could have the facet CARDINALITY defined for the ISBN slot, with facet value equal to "1". Own slots describe properties of specific individuals and other individuals from the same class may have different own slots.

Sometimes we have slots that describe properties valid for every individual of the class. These slots are called *template slots*, in the same way we define *template facets*. Template slots are inherited (i.e. every class has the template slots of its super class).

Classes, individuals, slots and facets are represented by *frames*.

OKBC defines only one standard slot : DOCUMENTATION which is intended to provide documentation for the class/individual. Slot values should be strings.

Some of the standard facets are :

- VALUE-TYPE
- CARDINALITY
- MINIMUM-CARDINALITY / MAXIMUM-CARDINALITY
- NUMERIC-MINIMUM / NUMERIC-MAXIMUM

- COLLECTION-TYPE

We will explain the semantics of the VALUE-TYPE facet, detailed explanations of other facets are available in the OKBC specification [2]. VALUE-TYPE specifies a type restriction on the values that could be assigned to a slot. For example if the slot S has class T for the facet VALUE-TYPE then if an individual Ind assigns some value to the slot S then this value is of type T.

Sometimes it is necessary to define slots with properties (facets) that hold (i.e. have the same value) for every frame (class or individual) that uses the slot. This is achieved with slots on slots, i.e. slots for slot frames.

The most important slot of this type is DOMAIN which specifies the classes that could use this slot. In other words if S is a slot and C is a class and the DOMAIN relation holds for S and C then :

1. If Ind is some individual that assigns value to the slot S then Ind must be of class C
2. If Csub is some class that assigns value to the slot S (i.e. S is template slot for Csub) then Csub is either subclass of C or C itself

The default value for DOMAIN is THING , i.e. if we do not explicitly specify the domain for the slot every class could use it.

Other slots that are defined for slot frames are :

- SLOT-VALUE-TYPE
- SLOT-CARDINALITY
- SLOT-MINIMUM-CARDINALITY / SLOT-MAXIMUM-CARDINALITY
- SLOT-NUMERIC-MINIMUM / SLOT-NUMERIC-MAXIMUM
- SLOT-COLLECTION-TYPE

We will explain only the meaning of SLOT-VALUE-TYPE using the VALUE-TYPE facet (mentioned above) because other slots could be explained in the same manner – if a slot S has class T for SLOT-VALUE-TYPE and C is a class so that DOMAIN holds for S and C, then all individuals of class C have T as a value for the facet VALUE-TYPE.

For example we may have two classes – BOOK1 and BOOK2 which have the ISBN template slot but BOOK1 specifies *string* for the VALUE-TYPE of the slot while BOOK2 specifies *number*. Then all the individuals of BOOK1 could assign only strings to ISBN while individuals of BOOK2 could assign only numbers to the slot. On the other hand if we have defined ISBN as a slot so that BOOK1 and BOOK2 are in its domain then if we set *string* for SLOT-VALUE-TYPE of ISBN, we will ensure that all individuals of BOOK1 and all individuals of BOOK2 could assign only *string*s to the ISBN slot.

So far we have described classes, individuals, slots, facets as defined by OKBC. We will show how XOL describes them in XML. Every XML document representing some ontology should conform to the XOL DTD (or the XSchema available in Appendix C). Every XOL document has the following parts:

- module section
- class section
- slot section
- individual section

The XOL document contains one and only one module, which identifies the ontology being described. All other sections are contained in the module section. Taking a look at the XOL DTD (or XSchema) we see that a module could look like:

```
<module>
        <name>Media ontology</name>
```

```
		<kb-type> some famous KBS</kb-type>
		<version>1.0</version>
		<documentation>Ontology for online books and videos </documentation>
</module>
```

Notice that according to the DTD (XML Schema) the root element of the module section could be one of the following elements (which are synonymous): module, ontology, kb, database, dataset.

Some comments about the module section – the root element should have a **name** element (the name of the ontology), the rest of the elements are optional, **kb-type** is intended to specify the KBS from which the ontology originally comes (if available), **version** and **documentation** are intended to provide additional information about the ontology.

Coming next is the class section. Here we describe the classes presented in our ontology, their relationships and template slots. So we could have something like:

```
<class>
		<name>MEDIA</name>
		<documentation>The class of all media types</documentation>
		<subclass-of>THING</subclass-of>
</class>

<class>
		<name>BOOK</name>
		<documentation>The class of all books</documentation>
		<subclass-of> MEDIA </subclass-of>
</class>

<class>
		<name>VIDEO</name>
		<documentation>The class of all videos</documentation>
		<subclass-of> MEDIA </subclass-of>
</class>

<class>
		<name>DVD</name>
		<documentation>The class of all DVDs</documentation>
		<subclass-of> MEDIA </subclass-of>
</class>

<class>
		<name>ORGANISATION</name>
		<documentation>The class of all organizations</documentation>
		<subclass-of> THING </subclass-of>
</class>

<class>
		<name>COMPANY</name>
		<documentation>The class of all companies</documentation>
		<subclass-of> ORGANISATION</subclass-of>
</class>
```

The next section is for slot definitions – i.e. we describe slot frames here. Every slot frame describes either a template slot or an owns slot and this is specified by the type attribute of the <slot> tag. Let us define the following slots for the book class: author, ISBN, title, publisher. We could have something like:

```
<slot type="own">
		<name>AUTHOR</name>
		<documentation>The name of the book/video author </documentation>
		<domain>BOOK</domain>
		<domain>VIDEO</domain>
		<slot-value-type>STRING</slot-value-type>
		<slot-minimum-cardinality>1</slot-minimum-cardinality >
		<slot-maximum-cardinality>3</slot- maximum -cardinality >
```

```
</slot>
```

Note that we allow individuals of BOOK and VIDEO to have author, while we do not allow it for DVD.

```
<slot type="own">
	<name>ISBN</name>
	<documentation>The ISBN of the book</documentation>
	<domain>BOOK</domain>
	<slot-value-type>STRING</slot-value-type>
	<slot-cardinality>1</slot-cardinality >
</slot>

<slot type="own">
	<name>PUBLISHER</name>
	<documentation>The publisher of the media</documentation>
	<domain>MEDIA</domain>
	<slot-value-type>COMPANY</slot-value-type>
	<slot-cardinality>1</slot-cardinality >
</slot>
```

The next section is for individuals:

```
<individual>
	<name>Knopf</name>
	<documentation>Knopf publishing company</documentation>
	<instance-of>COMPANY</instance-of>
</individual>


<individual>
	<name>Birdy</name>
	<documentation>cool book</documentation>
	<instance-of>BOOK</instance-of>

	<slot-values>
		<name>AUTHOR</name>
		<value>William Wharton</value>
	</slot-values>

	<slot-values>
		<name>ISBN</name>
		<value>0679734120</value>
	</slot-values>

	<slot-values>
		<name>PUBLISHER</name>
		<value>Knopf</value>
	</slot-values>
</individual>
```

The general rules that XOL documents should follow (as defined by the authors) are:

- names of classes/individuals/slots must be unique within a XOL file (i.e. ontology)
- super classes should be defined prior to their subclasses
- classes must be defined prior to their instances (this restriction is a bit redundant, having in mind that the XOL DTD demands that the class section be specified before the individual section)
- the identifier provided for the *instance-of* and *subclass-of* relations should be a name of a *class*
- the same as above for *slot-values* and *slot* respectively
- slots could be used only with classes and instances within their domain (remember the purpose of the DOMAIN slot)
- values of a slot should conform to the SLOT-VALUE-TYPE definition

- classes should be defined prior to their slots (redundancy again – take a look at the DTD)

We would like to add that the values for facets (slots) like CARDINALITY (SLOT-CARDINALITY) should be meaningful, i.e. non-negative numbers. This could be enforced with a XML Schema for XOL but not with a DTD.

In addition we could mention the following inconsistencies

- while OKBC states that a valid value for the facet COLLECTION-TYPE and the slot SLOT-COLLECTION-TYPE is one of *set, list, bag,* XOL defines the corresponding tags as having PCDATA values (i.e. any character value is valid) which may lead to confusion for the application processing the ontology expressed in XOL.

- XOL makes no difference between *primitive* and *non-primitive* classes – a difference that is stated in OKBC (non-primitive class is a class for which the template facet values and template slot values specify the *necessary* and the *sufficient* conditions for an instance to belong to the class, while the template facet/slot values for a primitive class specify only the *necessary* conditions, for example we could define TRIANGLE as a non-primitive class, subclass of POLYGON, with SLOT-CARDINALITY for the EDGE slot equal to “3”).

- the DTD mandates that the class section precedes the slot section which makes use of template slots inconsistent (the class will set value to a slot that is not defined yet) while this will not be a problem for own slots, because the slot section precedes the individuals section.

3.2.1 Links

1. XOL: An XML-Based Ontology Exchange Language (http://www.ai.sri.com/pkarp/xol/xol.html)
2. OKBC Specification (http://www.ai.sri.com/~okbc/spec.html)
3. OKBC information (http://www.ai.sri.com/~okbc/)

3.3 OIL – Ontology Interchange Language (version 1.0)

The Ontology Interchange Language was designed with XOL in mind but it is not just an extension of XOL. In fact there are many differences between the two languages. OIL authors have combined aspects from three different domains – the formal semantics from Description Logic, the frame based modeling primitives and the XML based syntax.

The OIL DTD and schema (as defined by the authors) is included in Appendix D.

OIL documents (i.e. ontologies expressed in OIL) consist of a container and definition sections. The container section is similar to the module section in XOL, while the definition section serves as the class and slot sections in XOL. There is no individuals section.

The purpose of the container section is to describe properties of an ontology, like name, author, subject, description, etc. i.e. the metadata about the ontology. The metadata is represented with the RDF vocabulary for the Dublin Core Metadata Element Set. An example container section could look like (take a look at the OIL DTD or schema):

```
<ontology-container>
```

```
<rdf:RDF  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:dc="http://purl.oclc.org/dc#"

          <rdf:Description about = "">
                    <dc:Title>Media Ontology</dc:Title>
                    <dc:Creator>unknown</dc:Creator>
                    <dc:Subject>books, videos, dvd</dc:Subject>
                    <dc:Description>Ontology about media </dc:Description>
                    <dc:Type>ontology</dc:Type>
                    <dc:Format>text/xml</dc:Format>
                    <dc:Indentifier>http://pillango.sirma.bg/sample-ontology.xml</dc:Indentifier>
                    <dc:Language>OIL</dc:Language>
          </rdf:Description>

</rdf:RDF>

</ontology-container>
```

The container section consists of the following elements (details about each element could be found in [2]): title, creator, subject, description, publisher, contributor, date, type, format, indentifier, source, language, relation, rights.

Unlike XOL in OIL individuals cannot be defined. Moreover OIL lacks any notion of primitive data types like number, string, integer and it supports only the *set* as a collection type (while OKBC defines set, list and bag). Slot definitions in OIL have some differences compared to the ones in XOL. For example facets like *numeric-minimum* and *numeric-maximum* are not supported.

Slots in OIL could be characterized as *symmetric* or *transitive* (for example we could define slot **is-ancestor** as transitive while **has-spouse** as symmetric). In addition slots could be organized in hierarchies with the *subslot-of* slot. Every slot has a unique *name* and optional *documentation*, *domain* (the classes whose individuals could assign values to the slot), *range* (classes which individuals could be slot values) and *properties* (transitive or symmetric). We could rewrite the slot definition from the XOL examples in OIL:

```
<ontology-definitions>

          <slot-def>
                    <slot name="AUTHOR"/>
                    <documentation>The name of the book/video author </documentation>
                    <domain>
                              <OR>
                                        <class name="BOOK" />
                                        <class name="VIDEO" />
                              </OR>
                    </domain>
          </slot-def>

          <slot-def>
                    <slot name="ISBN"/>
                    <documentation>The name of the book author </documentation>
                    <domain>
                              <class name="BOOK" />
                    </domain>
          </slot-def>

          <slot-def>
                    <slot name=" PUBLISHER "/>
                    <documentation> The publisher of the media </documentation>
                    <domain>
                              <class name="BOOK" />
                              <class name="VIDEO" />
                              <class name="DVD" />
```

```
		</domain>
	</slot-def>

	….
	other slot definitions
	….

	….
	class definitions
	….

</ontology-definitions>
```

Few important notes: the slot value for a domain is a *class expression* – a boolean combination of classes using the operators AND, OR and NOT. This feature comes from the Description Logic roots of OIL and cannot be seen in XOL (and OKBC). Another interesting fact is that in OIL the cardinality/value type/etc cannot be specified in the slot definition (as slots) but in the slot constraints in classes that use the slot (as facets).

A class definition consists of a class name, documentation, set of super class names and slot constraints. Unlike XOL, classes in OIL are either *primitive* or *defined* (non-primitive) with primitive as default. If a class is primitive, its definition (slot constraints and super class components) is necessary but not sufficient condition for an individual to be an instance of the class. On the opposite the definition of *defined* classes is both necessary and sufficient condition for membership of the class.

Our sample XOL ontology could be expressed in OIL like:

```
<ontology-definitions>

	….
	slot definitions
	….

	<class-def>
		<class name=”STRING” / >
		<documentation>strings</documentation>
	</class-def >


	<class-def>
		<class name=”MEDIA” / >
		<documentation>The class of all media types</documentation>
	</class-def >


	<class-def>
		<class name=”PERSON” / >
		<documentation>The class of all persons (used for the AUTHOR slot) </documentation>
	</class-def >


	<class-def>
		<class name=”MAN” / >
		<documentation>The class of all male humans</documentation>

		<subclass-of>
			<class name=”PERSON” />
		</subclass-of>
	</class-def >


	<class-def>
		<class name=”WOMAN” / >
		<documentation> The class of all female humans </documentation>
```

```
        <subclass-of>
                <AND>
                        <class name="PERSON">
                        <NOT>
                                <class name="MAN" />
                        </NOT>
                </AND>
        </subclass-of>
</class-def >


<class-def >
        <class name="ORGANISATION" />
        <documentation>The class of all orgs</documentation>
</class-def >


<class-def >
        <class name="COMPANY" />
        <documentation>The class of all companies</documentation>

        <subclass-of>
                <class name="ORGANISATION" />
        </subclass-of>
</class-def >


<class-def >
        <class name="BOOK" />
        <documentation>The class of all books</documentation>

        <subclass-of>
                <class name="MEDIA" />
        </subclass-of>

        <slot-constraint>
                <slot name="AUTHOR"/>

                <value-type>
                        <class name="PERSON" />
                </value-type>

                <max-cardinality>
                        <number>3<number>
                        <class name="PERSON">
                </max-cardinality>

                <min-cardinality>
                        <number>1<number>
                        <class name="PERSON">
                </min-cardinality>
        </slot-constraint>

        <slot-constraint>
                <slot name="ISBN"/>

                <value-type>
                        <class name="STRING" />
                </value-type>

                …
                other facets
                …

        </slot-constraint>

        <slot-constraint>
                <slot name="PUBLISHER"/>

                <value-type>
```

```
                    <class name=”COMPANY” />
                </value-type>

                …
                other facets
                ...
            </slot-constraint>

        </class-def >

        …
        other class definitions (VIDEO, DVD)
        …

<ontology-definitions>
```

Few points of interest:

1. We defined the class STRING (for ISBN slot value) because there are no primitive datatypes in OIL.
2. We added a class PERSON and defined two subclasses – MAN and WOMAN which are disjoint (take a look at the definition for WOMAN) . Such disjoint classes cannot be expressed in XOL (and OKBC)
3. OIL allows classes to be specified in the cardinality related facets which gives more expressive power (for example we can define some slot as having cardinality N for instances of class X and cardinality M for instances of class Y) . Of course this could lead to some inconsistencies.

The correspondences between XOL and OIL look like:

XOL notion	OIL notion
module/ontology/kb/database/dataset elements	*ontology-container* element
X	*relation* (references to other ontologies within *ontology-container*)
name (within *module*/… element)	*dc:Title* element within *ontology-container*
X	*import* element (ontologies to be included)
class element	*class-def* element
name (within class or slot)	*name* attribute of *class* or *slot* element
documentation (within *class* or *slot*)	*documentation* (within *class-def* or *slot-def*)
X	*type* attribute of *class-def* (primitive/defined)
subclass-of element	*subclass-of* element
X	Class expression
X	Disjoint classes
value (wihin *slot-values*)	*has-value* (within *slot-constraints*)
Primitive datatypes	X
slot element	*slot-def* element
domain element within *slot*	*domain* element within *slot-def*
slot-value-type slot	*range* (within *slot-def*) or *value-type* (within *slot-constraint*)
slot-inverse slot	*inverse* element within *slot-def*
X	*properties* element (within *slot-def*)
X	*subslot-of* (within *slot-def*)
individual element	X
instance-of element	X
max-cardinality within template slot	*max-cardinality* within *slot-constraint*

slot-max-cardinality within own slot	X
numeric-minimum / Slot-numeric-minimum	X
slot-collection-type/collection-type	X

A very important feature of OIL is that ontology definitions (slot and class definitions) could be mapped into axioms in the SHIQ description logic for which sound and complete reasoning could be performed with the help of the FaCT system. Details about the mapping are available in [2]. The OIL site also provides a tool for translating OIL ontologies into SHIQ format [5]

3.3.1 Links

1. D. Fensel et al.: OIL in a nutshell (http://www.cs.vu.nl/~ontoknow/oil/downl/oilnutshell.pdf)
2. I. Horrocks et al.: The Ontology Inference Layer OIL (http://www.cs.vu.nl/~dieter/oil/Tr/oil.pdf)
3. M.C.A. Klein et al.: The Relation between Ontologies and Schema-Languages: Translating OIL-Specifications to XML-Schema (http://www.cs.vu.nl/~mcaklein/papers/oil-xmls.pdf)
4. OIL information - http://www.ontoknowledge.org/oil/
5. Translator from XML OIL 1.0 to FaCT (http://www.ontoknowledge.org/oil/Oil2fact.zip)

3.4 OIL – Ontology Interchange Language (version 2.0)

This section will examine the new features in OIL 2.0 like "top" and "bottom" concepts, sets, predefined types, instances, etc.

This section is incomplete.

3.4.1 Links

1. OIL-Lite 2.0 (http://www.ontoknowledge.org/oil/syntax/OIL-Lite/)
2. OIL-Standard 2.0 (http://www.ontoknowledge.org/oil/syntax/OIL-Standard/)

3.5 OML – Ontology Markup Language

The Ontology Consortium is no longer developing the OML language. Instead a new framework and language – the Information Flow Framework is being developed (see next chapters).

3.5.1 Links

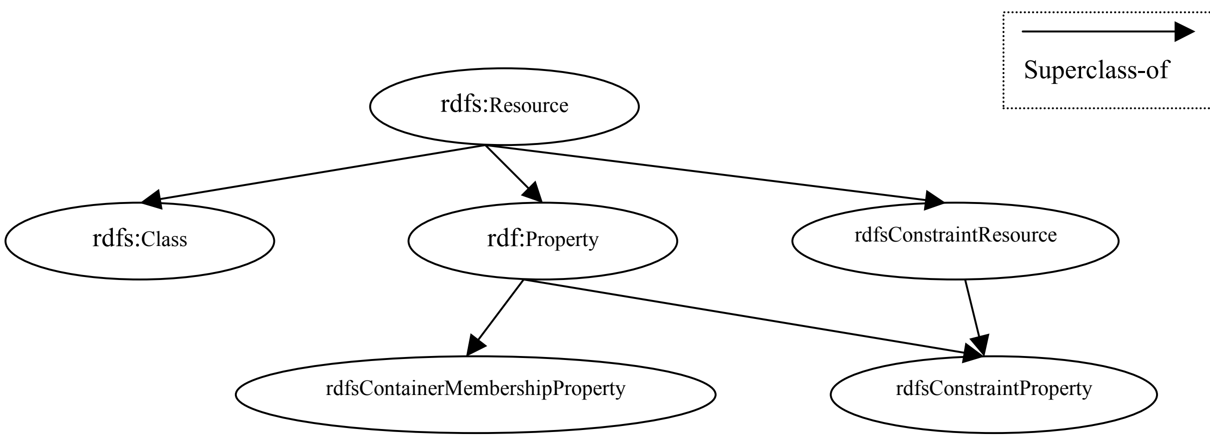
1. OML Information - http://www.ontologos.org/OML/OML%200.3.htm
2. IFF information - http://www.ontologos.org/IFF/The%20IFF%20Language.html

3.6 RDFS as Ontology Language

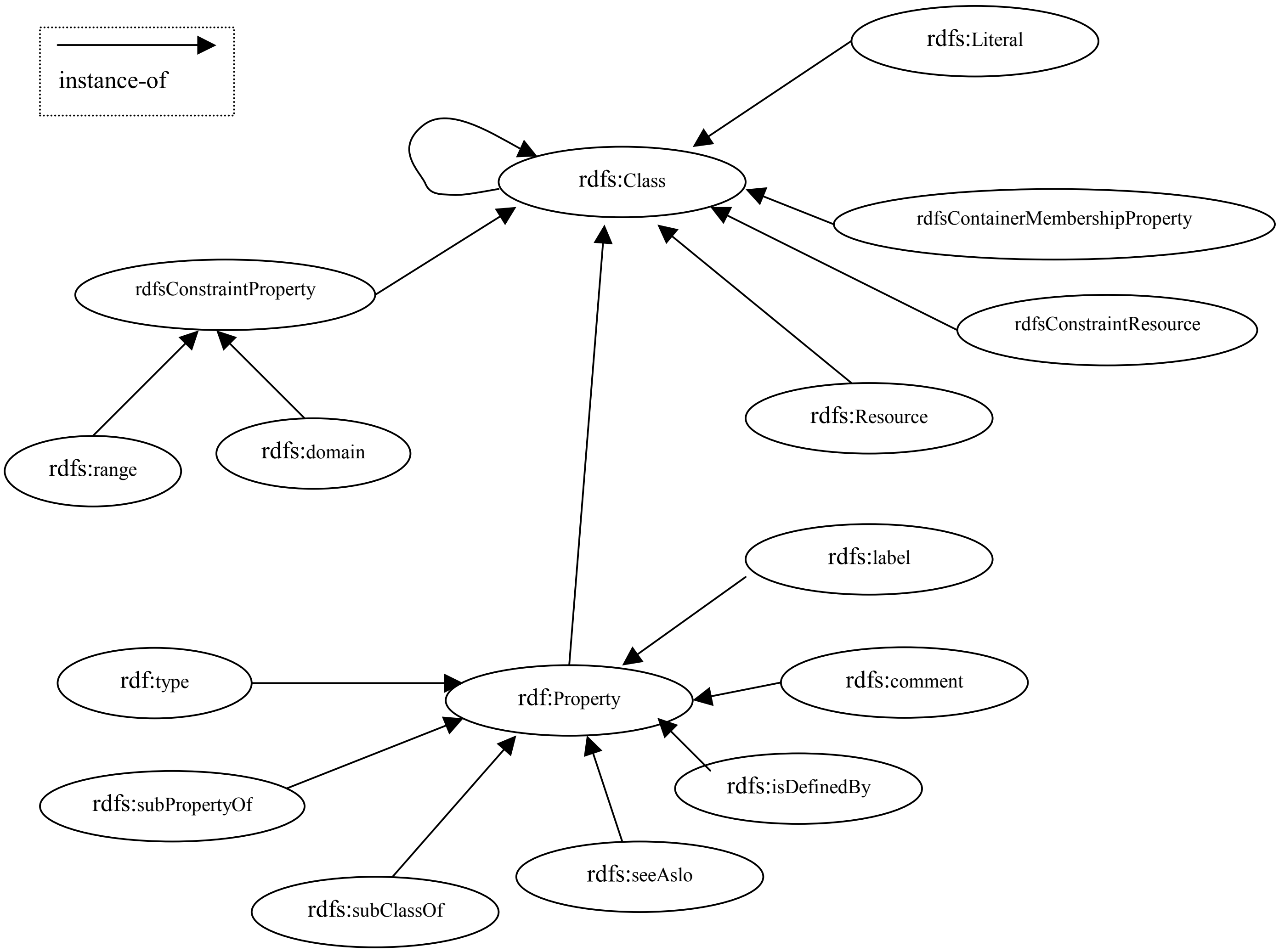
This section is based on [4] where the authors of OIL analyze the relation between OIL and RDFS.

We already gave a brief introduction to RDF – it describes metadata for resources on the web in means of statements, resources and properties. RDFS further extends RDF by adding more modeling primitives often found in ontology languages – classes, class inheritance, property inheritance, domain and range restrictions, ways to specify the class to which some instance belongs.

The superclass-of hierarchy in RDFS as defined in [1] looks like:



The instance-of relations as defined in [1] look like:



Note that there is no clear separation between the "rdf" and "rdfs" namespaces i.e. RDFS is not just an extension to RDF because RDF (rdf:Property) depends on RDFS too. Also note that RDFS defines no primitive datatypes except Literal.

The rdfs:Class is used to define classes/concepts (analogous to *class* elements in XOL and *class-def* elements on OIL). Classes are organized in hierarchy with the help of the rdfs:subClassOf property. RDFS has no notion of primitive/defined classes and complex class expression similar to the ones in OIL cannot be expressed directly. Resources (entities) that represent instances belong to some class and this is specified by the rdf:type property (analogous to *instance-of* elements in XOL).

Every class has associated with it a set of properties and constraints (which are properties too). Properties are of type rdf:Property (*slot/slot-def* in XOL/OIL). Properties could be organized in hierarchy with the help of rdfs:subPropertyOf property (*subslot-of* in OIL). Properties in RDFS are analogous to slots in XOL/OIL and they could be defined independently of classes.

One often criticized restriction for rdfs:subClassOf and rdfs:subPropertyOf is that they cannot form cycles in the class/slot inheritance graph thus equivalence between classes/slots cannot be expressed.

The standard constraints rdfs:range and rdfs:domain are used to specify the valid values for a property and the classes that may have the property respectively (compare with the *slot-value-type/range* and *domain* in XOL/OIL). The rdfs:range constraint should have at most one occurrence in the class definition. As there are no complex class expressions one way to specify more classes as valid values for a property is to create an artificial super class for them.

Let us define a simple class and its properties with RDFS:

```
<rdf:Property rdf:ID="CREATOR">
        <rdfs:comment>the agent who created the entity</rdfs:comment>
        <rdfs:range rdf:resource="#PERSON" />
        <!-- domain is unspecified -->
</rdf:Property>

<rdf:Property rdf:ID="AUTHOR">
        <rdfs:comment>The Author of the book/video </rdfs:comment>
        <rdfs:subPropertyOf rdf:resource="#CREATOR" />
        <rdfs:domain rdf:resource="#BOOK"/>
        <rdfs:domain rdf:resource="#VIDEO"/>
</rdf:Property>

<rdf:Property rdf:ID="PUBLISHER">
        <rdfs:comment>The publisher of the book/video/dvd</rdfs:comment>
        <rdfs:domain rdf:resource="#MEDIA"/>
        <rdfs:range rdf:resource="#ORGANIZATION" />
</rdf:Property>

<rdf:Property rdf:ID="ISBN">
        <rdfs:comment>The ISBN of the book </rdfs:comment>
        <rdfs:domain rdf:resource="#BOOK"/>
        <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal" / >
</rdf:Property>

….
```

The above section is analogous to the slot section in XOL and to the part of the ontology definitions section in OIL that contains the slot definitions. Now let us define some classes:

```
<rdfs:Class rdf:ID="MEDIA">
        <rdfs:comment> The class of all media types </rdfs:comment>
</rdfs:Class>


<rdfs:Class rdf:ID="PERSON">
        <rdfs:comment> The class of all persons </rdfs:comment>
</rdfs:Class>


<rdfs:Class rdf:ID="ORGANIZATION">
        <rdfs:comment> The class of all organizations </rdfs:comment>
</rdfs:Class>


<rdfs:Class rdf:ID="BOOK">
        <rdfs:comment> The class of all books </rdfs:comment>
        <rdfs:subClassOf rdf:resource="#MEDIA" />
</rdfs:Class>
```

```
<rdfs:Class rdf:ID=”VIDEO”>
        <rdfs:comment> The class of all videos </rdfs:comment>
        <rdfs:subClassOf rdf:resource=”#MEDIA” />
</rdfs:Class>


<rdfs:Class rdf:ID=”DVD”>
        <rdfs:comment> The class of all DVDs </rdfs:comment>
        <rdfs:subClassOf rdf:resource=”#MEDIA” />
</rdfs:Class>
```

…. Definitions for other classes like PERSON, ORGANIZATION ,…

The above section is analogous to the class section in XOL and to the part of the ontology definitions section in OIL that contains the class definitions. We could also define some individuals:

```
<rdf:Description rdf:ID=”Knopf”>
        <rdf:comment> Knopf publishing company </rdf:comment>
        <rdf:type rdf:ID=”#ORGANIZATION” />
</rdf:Description>


<rdf:Description rdf:ID=”Wharton”>
        <rdf:comment> William Wharton </rdf:comment>
        <rdf:type rdf:ID=”#PERSON” />
        <FIRST_NAME> William </FIRST_NAME>
        <LAST_NAME> Wharton </LAST_NAME>
</rdf:Description>


<rdf:Description rdf:ID=”Birdy”>
        <rdf:comment> cool book </rdf:comment>
        <rdf:type rdf:ID=”#BOOK” />
        <AUTHOR rdf:resource=”#Wharton” />
        <PUBLISHER rdf:resource=”#Knopf” />
        <ISBN>0679734120</ISBN>
</rdf:Description>
```

Note that RDFS specifies no structure about the document representing the ontology (i.e. there are no separate module/ontology-container and class/ontology-definition sections as in XOL/OIL). The Dublin Core element set could be used to form a description of the ontology (name, author, language, etc).

The correspondences between XOL and OIL look like:

XOL notion	OIL notion	RDFS notion
module/ontology/kb/database/dataset elements	*ontology-container* element	X
X	*relation* (references to other ontologies within *ontology-container*)	X
name (within *module*/*kb*/… element)	*dc:Title* element within *ontology-container*	X
X	*Import* element (ontologies to be included)	X
class element	*class-def* element	*rdfs:Class* element
name (within class or slot)	*name* attribute of *class* or *slot* element	*rdf:ID* attribute within *rdfs:Class*
documentation (within *class* or *slot*)	*Documentation* (within *class-def* or *slot-def*)	*rdfs:Comment* (within *rdfs:Class*)

X	*type* attribute of *class-def* (primitive/defined)	X
subclass-of element	*subclass-of* element	*Rdfs:subClassOf* element
X	Class expression	X
X	Disjoint classes	X
Primitive datatypes	X	X
slot element	*slot-def* element	*rdf:Property* element
domain element within *slot*	*domain* element within *slot-def*	*rdfs:domain* element within *rdf:Property*
slot-value-type slot	*range* (within *slot-def*) or *value-type* (within *slot-constraint*)	*rdfs:range* element within *rdf:Property*
slot-inverse slot	*inverse* element within *slot-def*	X
X	*properties* element (within *slot-def*)	X
X	*subslot-of* (within *slot-def*)	*rdfs:subPropertyOf* (within *rdfs:Property*)
Individual element	X	*rdf:Description*/*rdfs:Resource* element
Instance-of element	X	*rdf:type* element (within description)
Value (within *slot-values*)	*has-value* facet (within *slot-constraint*)	The content of the element representing the slot
max-cardinality within template slot	*max-cardinality* within *slot-constraint*	X
slot-max-cardinality within own slot	X	X
Numeric-minimum / *Slot-numeric-minimum*	X	X
slot-collection-type/collection-type	X	?

3.6.1 Links

1. RDFS Specification (http://www.w3.org/TR/PR-rdf-schema)
2. “RDF Tutorial” (http://www710.univ-lyon1.fr/~champin/rdf-tutorial/)
3. “An Extensible Approach for Modeling Ontologies in RDFS” (http://www.aifb.uni-karlsruhe.de/~sst/Research/Publications/onto-rdfs.pdf)
4. “Adding formal semantics to the Web: building on top of RDF Schema” (http://www.ontoknowledge.org/oil/extending-rdfs.pdf)
5. “RDF: A Frame System for the Web” (http://www.ltg.ed.ac.uk/~ht/ora-rdf-dagstuhl.html)
6. RDFSViz - visualization service for ontologies represented in RDF Schema (http://www.dfki.uni-kl.de/frodo/RDFSViz/)

3.7 IFF – The Information Flow Framework Language

IFF is a sequel of OML which is intended to “incorporate ideas from other knowledge representation languages such as RDFS, OKBC, OIL, CGIF and CycL”.

This section is incomplete.

3.7.1 Links

1. IFF information - http://www.ontologos.org/IFF/The%20IFF%20Language.html

4 Appendixes

4.1 Appendix A : List of all(?) acronyms

DOM - Document Object Model
DTD - Document Type Definition
IFF **-** Information Flow Framework
OIL **-** Ontology Interchange Language
OML **-** Ontology Markup Language
RDF(S) **-** Resource Description Framework (Schema)
SAX - Simple API for XML
SGML - Standard Generalized Markup Language
URI **-** Unifier Resource Identifier
XLink - eXtensible Linking Language
XML – eXtensible Markup language
XML-QL XML Query Language
XOL - XML-based Ontology exchange Language
XQL - eXtensible Query Language
XSL(T) **-** eXtensible Stylesheet Language (for Transformations)

4.2 Appendix B : XML tools

This section will describe various tools related to XML (XML parser, XSL processors and editors, XPath processors and editors, RDFS editors, XQL and Quilt query engines, XML aware ontology editors)

This section is incomplete

4.3 Appendix C : XOL DTD and schema

This is a modified DTD for XOL, the original one contains some minor errors

```
<!ELEMENT module (name, ( kb-type | db-type )?, package?, version?, documentation?, class*,  slot*, individual*) >
<!ELEMENT ontology (name, ( kb-type | db-type )?, package?, version?, documentation?, class*,  slot*, individual*)  >
<!ELEMENT database (name, ( kb-type | db-type )?, package?, version?, documentation?, class*,  slot*, individual*) >
<!ELEMENT kb (name, ( kb-type | db-type )?, package?, version?, documentation?, class*,  slot*, individual*) >
<!ELEMENT dataset (name, ( kb-type | db-type )?, package?, version?, documentation?, class*,  slot*, individual*)  >

<!ELEMENT db-type (#PCDATA)>
<!ELEMENT package (#PCDATA)>
<!ELEMENT version (#PCDATA)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT kb-type (#PCDATA)>
<!ELEMENT class  (name, documentation?, ( subclass-of | instance-of | slot-values)* ) >
<!ELEMENT documentation (#PCDATA)>
<!ELEMENT subclass-of (#PCDATA)>
<!ELEMENT instance-of (#PCDATA)>
<!ELEMENT slot    (name, documentation?, ( domain |
```

```
                                        slot-value-type |
                                        slot-inverse |
                                        slot-cardinality |
                                        slot-maximum-cardinality |
                                        slot-minimum-cardinality |
                                        slot-numeric-minimum |
                                        slot-numeric-maximum |
                                        slot-collection-type |
                                        slot-values )* ) >

<!ATTLIST slot          type ( template | own ) "own" >

<!ELEMENT individual (name, documentation?, instance-of* , slot-values*) >
<!ELEMENT type (#PCDATA)>
<!ELEMENT domain (#PCDATA)>
<!ELEMENT slot-value-type (#PCDATA)>
<!ELEMENT slot-inverse (#PCDATA)>
<!ELEMENT slot-cardinality (#PCDATA)>
<!ELEMENT slot-maximum-cardinality (#PCDATA)>
<!ELEMENT slot-minimum-cardinality (#PCDATA)>
<!ELEMENT slot-numeric-minimum (#PCDATA)>
<!ELEMENT slot-numeric-maximum (#PCDATA)>
<!ELEMENT slot-collection-type (#PCDATA)>
<!ELEMENT slot-values     (name, value*, (facet-values |
                                        value-type |
                                        inverse |
                                        cardinality |
                                        maximum-cardinality |
                                        minimum-cardinality |
                                        numeric-minimum |
                                        numeric-maximum |
                                        some-values |
                                        collection-type |
                                        documentation-in-frame)*)>

<!ELEMENT facet-values (name, value*)>
<!ELEMENT value   (#PCDATA)>
<!ELEMENT value-type (#PCDATA)>
<!ELEMENT inverse (#PCDATA)>
<!ELEMENT cardinality (#PCDATA)>
<!ELEMENT maximum-cardinality (#PCDATA)>
<!ELEMENT minimum-cardinality (#PCDATA)>
<!ELEMENT numeric-minimum (#PCDATA)>
<!ELEMENT numeric-maximum (#PCDATA)>
<!ELEMENT some-values (#PCDATA)>
<!ELEMENT collection-type (#PCDATA)>
<!ELEMENT documentation-in-frame (#PCDATA)>
```

This is XML Schema for XOL:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <xsd:complexType name="facet-valuesType" content="elementOnly">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="value" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="individualType" content="elementOnly">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
```

```
            <xsd:element name="documentation" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="instance-of" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="slot-values" type="slot-valuesType" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="slot-valuesType" content="elementOnly">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="value" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="facet-values" type="facet-valuesType"/>
                <xsd:element name="value-type" type="xsd:string"/>
                <xsd:element name="inverse" type="xsd:string"/>
                <xsd:element name="cardinality" type="xsd:string"/>
                <xsd:element name="maximum-cardinality" type="xsd:string"/>
                <xsd:element name="minimum-cardinality" type="xsd:string"/>
                <xsd:element name="numeric-minimum" type="xsd:string"/>
                <xsd:element name="numeric-maximum" type="xsd:string"/>
                <xsd:element name="some-values" type="xsd:string"/>
                <xsd:element name="collection-type" type="xsd:string"/>
                <xsd:element name="documentation-in-frame" type="xsd:string"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="slotType" content="elementOnly">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="documentation" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="domain" type="xsd:string"/>
                <xsd:element name="slot-value-type" type="xsd:string"/>
                <xsd:element name="slot-inverse" type="xsd:string"/>
                <xsd:element name="slot-cardinality" type="xsd:string"/>
                <xsd:element name="slot-maximum-cardinality" type="xsd:string"/>
                <xsd:element name="slot-minimum-cardinality" type="xsd:string"/>
                <xsd:element name="slot-numeric-minimum" type="xsd:string"/>
                <xsd:element name="slot-numeric-maximum" type="xsd:string"/>
                <xsd:element name="slot-collection-type" type="xsd:string"/>
                <xsd:element name="slot-values" type="slot-valuesType"/>
            </xsd:choice>
        </xsd:sequence>

        <xsd:attribute name="type" use="default" value="own">
            <xsd:simpleType base="xsd:string">
                <xsd:enumeration value="template"/>
                <xsd:enumeration value="own"/>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>

    <xsd:complexType name="classType" content="elementOnly">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="documentation" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="subclass-of" type="xsd:string"/>
                <xsd:element name="instance-of" type="xsd:string"/>
                <xsd:element name="slot-values" type="slot-valuesType"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="KBType" content="elementOnly">
        <xsd:sequence>
```

```
        <xsd:element name="name" type="xsd:string"/>
            <xsd:choice minOccurs="0" maxOccurs="1">
                <xsd:element name="kb-type" type="xsd:string"/>
                <xsd:element name="db-type" type="xsd:string"/>
            </xsd:choice>
            <xsd:element name="package" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="version" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="documentation" type="xsd:string" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="class" type="classType" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="slot" type="slotType" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="individual" type="individualType" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>


    <xsd:element name="module" type="KBType" />
    <xsd:element name="ontology" type="KBType" />
    <xsd:element name="database" type="KBType" />
    <xsd:element name="kb" type="KBType" />
    <xsd:element name="dataset" type="KBType" />

</xsd:schema>
```

4.4 Appendix D : OIL DTD and schema

The DTD (http://www.ontoknowledge.org/oil/dtd/) and XML Schema (http://www.ontoknowledge.org/oil/xml-schema/) are copied from the OIL site.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for Ontology Integration Language OIL -->
<!-- version 01 May 2000 -->

<!ELEMENT ontology (ontology-container, ontology-definitions)>

<!-- Ontology container -->
<!ELEMENT ontology-container          (rdf:RDF)>
<!-- This part contains meta-data about the ontology.  It is formatted according [Miller et al., 1999] -->

<!ELEMENT       rdf:RDF (rdf:Description)>
<!ATTLIST       rdf:RDF
                        xmlns:rdf CDATA #FIXED "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                        xmlns:dc  CDATA #FIXED "http://purl.oclc.org/dc#"
                        xmlns:dcq CDATA #FIXED "http://purl.org/dc/qualifiers/1.0/">
<!ELEMENT       rdf:Description (   (dc:Title+, dc:Creator+, dc:Subject*, dc:Description+,
                                    dc:Publisher*, dc:Contributor*, dc:Date*, dc:Type+,
                                    dc:Format*, dc:Identifier+, dc:Source*,
                                    dc:Language+, dc:Relation*, dc:Rights*) |
                                (dcq:descriptionType, rdf:value) |
                                (dcq:relationType, rdf:value) )>
<!ATTLIST       rdf:Description         about CDATA #IMPLIED >
<!ELEMENT       dc:Title (#PCDATA)>
<!ELEMENT       dc:Creator (#PCDATA)>
<!ELEMENT       dc:Subject (#PCDATA)>
<!ELEMENT       dc:Description (#PCDATA | rdf:Description)*>
<!ELEMENT       dc:Publisher (#PCDATA)>
<!ELEMENT       dc:Contributor (#PCDATA)>
<!ELEMENT       dc:Date (#PCDATA)>
<!ELEMENT       dc:Type (#PCDATA)>
<!ELEMENT       dc:Format (#PCDATA)>
<!ELEMENT       dc:Identifier (#PCDATA)>
<!ELEMENT       dc:Source (#PCDATA)>
```

```
<!ELEMENT        dc:Language (#PCDATA)>
<!ELEMENT        dc:Relation (#PCDATA | rdf:Description)*>
<!ELEMENT        dc:Rights (#PCDATA)>
<!ELEMENT        dcq:descriptionType (#PCDATA)>
<!ELEMENT        dcq:relationType (#PCDATA)>
<!ELEMENT        rdf:value (#PCDATA)>

<!-- Ontology-definitions -->
<!ELEMENT        ontology-definitions (imports?,
                                      rule-base?,
                                      (class-def | slot-def)* )>

<!-- Import-section with URI's to other ontology-files -->
<!ELEMENT        imports    (URI)+>
<!ELEMENT        URI        (#PCDATA)>

<!-- Rules with URL to definition -->
<!ELEMENT        rule-base (#PCDATA)>
<!ATTLIST        rule-base            type CDATA #REQUIRED>

<!-- Class-expressions -->
<!ENTITY         % class-expr "( class | slot-constraint | AND | OR | NOT)">
<!ELEMENT        AND    (   (%class-expr;),
                            (%class-expr;)+)>
<!ELEMENT        OR     (   (%class-expr;),
                            (%class-expr;)+)>
<!ELEMENT        NOT    (%class-expr;)>

<!-- Class-definition -->
<!ELEMENT        class-def  (class,
                            documentation?,
                            subclass-of?,
                            slot-constraint*)>
<!ATTLIST        class-def            type ( primitive | defined ) "primitive">

<!-- Class-name -->
<!ELEMENT        class      EMPTY>
<!ATTLIST        class      name CDATA #REQUIRED>
<!ELEMENT        documentation (#PCDATA)>
<!ELEMENT        subclass-of      (%class-expr;)+>

<!-- Slot-definition -->
<!ELEMENT        slot-def   (slot,
                            documentation?,
                            subslot-of?,
                            domain?,
                            range?,
                            inverse?,
                            properties?)>

<!-- Slot-name -->
<!ELEMENT        slot       EMPTY>
<!ATTLIST        slot       name CDATA #REQUIRED>
<!ELEMENT        subslot-of           (slot)+>
<!ELEMENT        domain     (%class-expr;)+>
<!ELEMENT        range      (%class-expr;)+>
<!ELEMENT        inverse    (slot)>

<!-- Slot-properties -->
<!ELEMENT        properties           ( transitive | symmetric | other )*>
<!ELEMENT        transitive  EMPTY>
<!ELEMENT        symmetric EMPTY>
<!ELEMENT        other      (#PCDATA)>

<!-- Slot-constraint -->
<!ELEMENT        slot-constraint (slot,
                                 (has-value | value-type | cardinality | max-cardinality | min-cardinality )+ )>
<!ELEMENT        has-value     (%class-expr;)+>
<!ELEMENT        value-type    (%class-expr;)+>
<!ELEMENT        cardinality    (number, %class-expr;)+>
```

```
<!ELEMENT    max-cardinality    (number, %class-expr;)+>
<!ELEMENT    min-cardinality    (number, %class-expr;)+>
<!ELEMENT    number             (#PCDATA)>
```

This is the XML Schema for OIL:

```
<schema xmlns="http://www.w3.org/1999/XMLSchema">
   <annotation>
      <documentation>
         XML Schema (Apr 07, 2000 WD) for the Ontology Inference Layer (OIL)
         http://www.ontoknowledge.org/oil/
         $Id: oil-schema.xml,v 2.3 2000/05/09 16:00:50 mcaklein Exp $
      </documentation>
   </annotation>

   <element name="ontology">
      <complexType content="elementOnly">
         <sequence>
            <element ref="ontology-container"/>
            <element ref="ontology-definitions"/>
         </sequence>
      </complexType>
   </element>

   <element name="ontology-container">
      <complexType content="elementOnly">
         <element ref="rdf:RDF"/>
      </complexType>
   </element>

   <element name="rdf:RDF">
      <complexType content="elementOnly">
         <element ref="rdf:Description"/>
         <attribute name="xmlns:dcq" type="string" value="http://purl.org/dc/qualifiers/1.0/" use="fixed"/>
         <attribute name="xmlns:rdf" type="string" value="http://www.w3.org/1999/02/22-rdf-syntax-ns#" use="fixed"/>
         <attribute name="xmlns:dc" type="string" value="http://purl.oclc.org/dc#" use="fixed"/>
      </complexType>
   </element>

   <element name="rdf:Description">
      <complexType content="elementOnly">
         <choice>
            <sequence>
               <element name="dc:Title" maxOccurs="unbounded" type="string"/>
               <element name="dc:Creator" maxOccurs="unbounded" type="string"/>
               <element name="dc:Subject" maxOccurs="unbounded" minOccurs="0" type="string"/>
               <element name="dc:Description" maxOccurs="unbounded" type="string">
                  <complexType content="mixed">
                     <group>
                        <sequence>
                           <element name="dcq:descriptionType">
                              <simpleType base="string">
                                 <enumeration value="version"/>
                              </simpleType>
                           </element>
                           <element name="rdf:value" type="string"/>
                        </sequence>
                     </group>
                  </complexType>
               </element>
               <element name="dc:Publisher" maxOccurs="unbounded" minOccurs="0" type="string"/>
               <element name="dc:Contributor" maxOccurs="unbounded" minOccurs="0" type="string"/>
               <element name="dc:Date" maxOccurs="unbounded" minOccurs="0" type="string"/>
               <element name="dc:Type" maxOccurs="unbounded" type="string"/>
```

```
                    <element name="dc:Format" maxOccurs="unbounded" minOccurs="0" type="string"/>
                    <element name="dc:Identifier" maxOccurs="unbounded"/>
                    <element name="dc:Source" maxOccurs="unbounded" minOccurs="0" type="string"/>
                    <element name="dc:Language" maxOccurs="unbounded" type="string"/>
                    <element name="dc:Relation" maxOccurs="unbounded" minOccurs="0">
                        <complexType content="mixed">
                            <group maxOccurs="unbounded" minOccurs="0">
                                <sequence>
                                    <element name="dcq:relationType">
                                        <simpleType base="string">
                                            <enumeration value="partOf"/>
                                        </simpleType>
                                    </element>
                                    <element name="rdf:value" type="string"/>
                                </sequence>
                            </group>
                        </complexType>
                    </element>
                    <element name="dc:Rights" maxOccurs="unbounded" minOccurs="0" type="string"/>
                </sequence>
            </choice>
            <attribute name="about" type="string" use="required"/>
        </complexType>
    </element>

    <element name="ontology-definitions">
        <complexType content="elementOnly">
            <sequence>
                <element ref="imports" minOccurs="0" maxOccurs="1"/>
                <element ref="rule-base" minOccurs="0" maxOccurs="1"/>
                <group minOccurs="0" maxOccurs="unbounded">
                    <choice>
                        <element ref="class-def"/>
                        <element ref="slot-def"/>
                    </choice>
                </group>
            </sequence>
        </complexType>
    </element>

    <element name="imports">
        <complexType content="elementOnly">
            <element name="URI" maxOccurs="unbounded" type="string"/>
        </complexType>
    </element>

    <element name="rule-base">
        <complexType content="textOnly">
            <attribute name="type" use="required" type="string"/>
        </complexType>
    </element>


    <complexType name="multiple-class-expr" content="elementOnly">
        <group>
            <choice>
                <element ref="class"/>
                <element ref="slot-constraint"/>
                <element name="AND" type="multiple-class-expr"/>
                <element name="OR" type="multiple-class-expr"/>
                <element name="NOT" type="class-expr"/>
            </choice>
        </group>
    </complexType>

    <complexType name="class-expr" base="multiple-class-expr" derivedBy="restriction">
```

```
    <group maxOccurs="1"/>
</complexType>

<complexType name="cardinality-expr" base="multiple-class-expr" derivedBy="extension">
    <element name="number" type="positive-integer"/>
</complexType>


<element name="class-def">
    <complexType content="elementOnly">
        <sequence>
            <element ref="class"/>
            <element ref="documentation" minOccurs="0" maxOccurs="1"/>
            <element ref="subclass-of" minOccurs="0" maxOccurs="1"/>
            <element ref="slot-constraint" maxOccurs="unbounded" minOccurs="0"/>
        </sequence>
        <attribute name="type" type="string" value="primitive" use="default">
            <simpleType base="string">
                <enumeration value="primitive"/>
                <enumeration value="defined"/>
            </simpleType>
        </attribute>
    </complexType>
</element>

<element name="class">
    <complexType content="empty">
        <attribute name="name" type="string" use="required"/>
    </complexType>
</element>

<element name="documentation">
    <simpleType name="documentation" base="string"/>
</element>

<element name="subclass-of" type="class-expr"/>

<element name="slot-def">
    <complexType content="elementOnly">
        <sequence>
            <element ref="slot"/>
            <element ref="documentation" minOccurs="0" maxOccurs="1"/>
            <element ref="subslot-of" minOccurs="0" maxOccurs="1"/>
            <element ref="domain" minOccurs="0" maxOccurs="1"/>
            <element ref="range" minOccurs="0" maxOccurs="1"/>
            <element ref="inverse" minOccurs="0" maxOccurs="1"/>
            <element ref="properties" minOccurs="0" maxOccurs="1"/>
        </sequence>
    </complexType>
</element>

<element name="slot">
    <complexType content="empty">
        <attribute name="name" type="string" use="required"/>
    </complexType>
</element>

<element name="subslot-of">
    <complexType content="elementOnly">
        <element ref="slot" maxOccurs="unbounded"/>
    </complexType>
</element>

<element name="domain" type="class-expr"/>

<element name="range" type="class-expr"/>
```

```
    <element name="inverse">
        <complexType content="elementOnly">
            <element name="slot"/>
        </complexType>
    </element>

    <element name="properties">
        <complexType content="elementOnly">
            <group maxOccurs="unbounded" minOccurs="0">
                <choice>
                    <element name="transitive">
                        <complexType content="empty"/>
                    </element>
                    <element name="reflexive">
                        <complexType content="empty"/>
                    </element>
                    <element name="symmetric">
                        <complexType content="empty"/>
                    </element>
                    <element name="other">
                        <simpleType name="other" base="string"/>
                    </element>
                </choice>
            </group>
        </complexType>
    </element>

    <element name="slot-constraint">
        <complexType content="elementOnly">
            <sequence>
                <element ref="slot"/>
                <group maxOccurs="unbounded">
                    <choice>
                        <element ref="value"/>
                        <element ref="value-type"/>
                        <element ref="cardinality"/>
                        <element ref="max-cardinality"/>
                        <element ref="min-cardinality"/>
                    </choice>
                </group>
            </sequence>
        </complexType>
    </element>

    <element name="value" type="class-expr"/>

    <element name="value-type" type="class-expr"/>

    <element name="cardinality" type="cardinality-expr"/>

    <element name="max-cardinality" type="cardinality-expr"/>

    <element name="min-cardinality" type="cardinality-expr"/>

</schema>
```