XOL: An XML-Based Ontology Exchange Language

Peter D. Karp[+], Vinay K. Chaudhri* and Jerome Thomere*

[+]Pangea Systems Inc.
4040 Campbell Ave.
Menlo Park, CA 94025
pkarp@PangeaSystems.com
And
***Artificial Intelligence Center**
SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
{chaudhri,thomere}@ai.sri.com

Version 0.5
February 17, 2000

1. Introduction

XOL provides a format for exchanging ontology definitions among a set of interested parties. The definitions that XOL is designed to encode include both schema information (meta-data), such as class definitions from object databases, and nonschema information (ground facts), such as object definitions from object databases.

The syntax of XOL is based on XML [GQ99] (http://www.w3.org/TR/REC-xml), which is a language for authoring documents for the WWW. XML syntax was chosen because it is reasonably simple to parse, its syntax is well defined, it is human readable, it appears that XML will be very widely used, and it also appears that many software tools for parsing and manipulating XML will soon be available. The semantics of XOL are based on OKBC-Lite, which is a simplified form of the knowledge model for OKBC (Open Knowledge Base Connectivity) [CF97,CF98] (http://www.ai.sri.com/~okbc/). OKBC is an API (application program interface) for accessing frame knowledge representation systems. Its knowledge model supports features most commonly found in knowledge representation systems, object databases, and relational databases (although not all primitive datatypes for object and relational databases are currently supported by XOL). OKBC-Lite extracts most of the essential features of OKBC, while not including some of its more complex aspects.

XOL is similar to other past ontology-exchange languages; its development was inspired by Ontolingua [Gr93] (http://www-ksl-svc.stanford.edu:5915/&service=frame-editor) and OML (http://wave.eecs.wsu.edu/CKRMI/OML.html). XOL differs from Ontolingua in having an XML-based syntax rather than a Lisp-based syntax; the semantics of OKBC-Lite are extremely similar to the semantics of Ontolingua. XOL differs from OML in that the semantics of OML are based on Conceptual Graphs, which have a number of differences from OKBC-Lite.

This introductory chapter describes the motivations for the language, as well as its intended uses. It also discusses tradeoffs considered during the design of the language.

1.1. Terminology

One definition of an ontology is that it is a specification of a conceptualization that is designed for reuse across multiple applications [Gr93,Gu95]. By a conceptualization, we mean a set of concepts, relations, objects, and constraints that define a semantic model of some domain of interest. An ontology is a specification of a conceptualization in the sense that it is a formal encoding of the concepts, relations, objects, and constraints within that semantic model.

The OKBC-Lite knowledge model provides precise definitions for what we mean by concepts, relations, objects, and constraints. The OKBC-Lite model is able to express the schemas used in a number of types of information management systems, such as relational database schemas, object database schemas, and knowledge bases used in frame knowledge representation systems. OKBC-Lite also subsumes less complex types of ontologies, including controlled vocabularies and taxonomies.

Note that as well as including schema information, our notion of ontology also includes nonschema information, such as individual objects within an object database manager, or individual rows within a relational database. Including both schema and data within the definition of ontology is not unusual given the central importance of reuse to the development of ontologies. It is common for application developers to want to reuse data as well as schema information across applications.

1.2. Motivations

Ontology sharing is important for several reasons. First, the ontology development that is a prerequisite for database design and for development of knowledge-based systems is difficult and time consuming. Different groups or organizations who wish to develop databases for the same types of information will often arrive at a solution faster by adopting an existing ontology than by developing a new ontology *de novo*.

Second, if several different databases that cover the same types of data (e.g., employee records) employ the same ontology, they simplify the problem of database integration, that is, of processing queries across multiple databases. The existence of different ontologies for the same types of data creates a semantic mismatch problem that complicates the multidatabase query problem.

Third, the developers of many databases (such as scientific databases) wish to make their schemas available to their user communities in a formally specified, readily understandable form, so that the users have a full understanding of the semantics of these databases.

Fourth, ontology sharing is important because ontologies themselves constitute a form of knowledge that some communities (such as scientific disciplines) wish to share.

The XOL language is designed to provide a mechanism for encoding ontologies within a flat file that may be easily published on the WWW for exchange among a set of application developers. The language is designed to be readable by humans, and to be easily parsable by programs of modest complexity. It is designed to be easily understandable by people who do not have a complete understanding of XML. It is also designed to be expressive, so that it may capture a rich variety of ontologies.

1.3. Intended Uses of XOL

XOL is a language for exchange of ontologies. By exchange language, we mean that XOL is intended to be used as an intermediate language for transferring ontologies among different database systems, ontology-development tools, or application programs.

For example, a group developing a scientific database might use the Oracle DBMS to actually implement the DB. However, that group could translate the DBMS schema from SQL into XOL, and then publish the resulting file on the WWW for reference by users of the database, or by other groups who are developing similar databases. The group might convert its schema into XOL, using an existing SQL-to-XOL translator. Or, if that group used an object-level graphical tool to design its DBMS schema, such as a UML-based tool, it might employ a pre-existing UML-to-XOL translator to translate the schema into XOL.

Another scientific database group might find the above ontology in its XOL form on the WWW. The group might employ another translator program to convert the ontology into the form used by its preferred DBMS or ontology-development tool. For example, it might convert the XOL ontology into a form usable by the GKB-Editor tool, such as the Ocelot frame representation system [KC99]. Once in that form, the ontology can be viewed or modified using the GKB-Editor.

As well as being used to exchange database schemas, XOL can be used to exchange database instance information. For example, all object instances within an object-oriented database, or a relational database, could be dumped to an XOL file for transfer to another organization that used either the same DBMS as the original organization, or a different DBMS. Again, translators between each DBMS and the XOL language must be written for the exchange to occur.

XOL is not a software environment for developing ontologies, such as Ontolingua or the GKB-Editor (http://www.ai.sri.com/~gkb/). However, since XOL files are textual, a text editor or XML editor may be used to author XOL files. Although some XOL users may use this approach, this specification discourages that practice because manual authoring of XOL files is likely to lead to files that are syntactically or semantically malformed, or both. However, since the XML language is in a very early stage of development, the expectation is that tools will soon be available so that hand crafting XOL documents will be necessary only for the near term.

1.4. Overview of XML

XML is a document markup language designed for constructing documents for the WWW. It is designed to be more flexible and powerful than is the HTML language. XML is a simplified version of the SGML markup language.

When we say XML is a document markup language, we mean that XML documents contain characters that encode the text of a document, plus characters that define the structure of the document, and that provide meta-information about the document.

In the XML fragment below, the text "<class>" is called an XML *start-tag*, and is part of the document markup. The text "</class>" is called an XML *end-tag*. Together, the start-tag and the end-tag define an *element* whose name is "class." The text "person" is part of the document text.

```
<class>
  <name>person</name>
</class>
```

The element names that may used in a particular family of XML documents, and the allowed nesting of those elements, are defined by a separate file called a DTD (Document Type Declaration).

1.5. The Generic Encoding Approach

The design of XOL deliberately uses what we call a generic approach to defining ontologies, meaning that the single set of XML tags (defined by a single XML DTD) defined for XOL can describe any and every ontology. This approach contrasts with the approaches taken by other XML schema languages, in which typically a generic set of tags is used to define the schema portion of the ontology, and the schema itself is used to generate a second set of application-specific tags (and an application-specific DTD) that in turn are used to encode a separate XML file that contains the data portion of the ontology.

This section describes this distinction in more detail, and explains the advantages and disadvantages of each approach.

Consider the following XOL definitions:

```
<class>
  <name>person</name>
</class>

<slot>
  <name>age</name>
  <domain>person</domain>
  <value-type>integer</value-type>
  <numeric-max>150</numeric-max>
</slot>

<individual>
  <name>fred</name>
  <type>person</type>
  <slot-values>
    <name>age</name>
    <value>35</value>
  </slot-values>
</individual>
```

All the XML elements of this specification (meaning all the words inside angle brackets), such as "class," "individual," and "name," are generic – they do not pertain to the Person ontology defined in this example – they pertain to all ontologies. *All* the ontology-specific information is in the text portion of the XML file, that is, between the pairs of elements.

In contrast, we might imagine using the following sort of XML markup to define the individual Fred:

```
<person>
  <name>fred</name>
  <age>35</age>
</person>
```

The preceding text is *not* a legal XOL specification because the tags it uses (such as "person" and "age") are not defined by XOL. This text is legal XML, but it is not legal XOL – not all legal XML is legal XOL. Legal XOL specifications must comply with the XOL DTD defined at http://www.w3.org/XOL/.

What are the relative advantages and disadvantages of the generic approach taken by XOL relative to the nongeneric approach?

The primary disadvantage of the generic approach is that XML parsing engines can perform only very limited types of checking on XOL specifications. For example, although the Person ontology at the beginning of this section clearly specifies that the Age slot is integer valued, that information is not captured within the XML DTD for XOL, and therefore the XML parsing engine cannot check that the value specified for Fred's age is in fact an integer. The XML parsing engine also cannot check that Age is in fact a valid slot for instances of the class Person, nor can it verify that the value 35 is smaller than the numeric maximum specified for Age. The XML engine cannot perform these checks because the only way to communicate the information required for those checks to the XML engine is via definitions within a DTD – whereas we have chosen to put those definitions in the XOL file itself. Therefore, applications that load XOL files are responsible for checking the semantic validity of XOL files.

The primary advantage of the XOL approach is simplicity. Only one XML DTD need be defined to describe any and every ontology. Using the nongeneric approach, every ontology must define a second, ontology-specific DTD for describing the data elements of the ontology, which essentially requires that any person using XOL must become familiar with DTDs, which can be fairly complex. Furthermore, rules would have to be defined that describe exactly how that second DTD is derived from the schema portion of the ontology, and most likely, programs would have to be written to generate such DTDs from schema specifications. The XML language provides no formal machinery to define those rules.

A second advantage of the generic approach is that it enables the construction of reusable software tools for manipulating XOL ontologies. For example, a translator that can convert an XOL ontology into some other form (such as an object-database schema) will work for *any* XOL ontology in any domain, be it an ontology of people, cars, or construction machinery. The reason is that XOL specifies a very controlled, restricted set of XML documents. In contrast, working with "raw XML," there are so many ways to encode the same information that it will prove impossible to write reusable software tools to operate on any XML data encodings. For example, consider the following alternative XML encodings of what is essentially the same underlying information:

```
<person>
  <name>fred</name>
  <age>35</age>
  <brother>bill</brother>
  <brother>harold</brother>
</person>

<person name=``fred''
   <age>35</age>
   <brother>bill,harold</brother>
</person>

<person  name=``fred''
         age=``35''
         brother=``bill,harold''
</person>
```

2. The OKBC-Lite Knowledge model

The OKBC-Lite knowledge model is an abridged version of the knowledge model used for the Open Knowledge Base Connectivity API. OKBC-Lite plays a central role in XOL by defining the semantic framework in which XOL ontologies are defined. The ontology building blocks defined by the OKBC-Lite Knowledge Model include classes, individuals, slots, facets, and knowledge bases. The knowledge model also recognizes some basic data types. We describe each of these constructs in the following sections.

The description of the OKBC knowledge model may be found elsewhere [CF97]. Although many aspects of the OKBC-Lite knowledge model are derived directly from the OKBC model, we have made several simplifications to the OKBC model. We have eliminated all references to the notion of "frame" because it does not have a clear portable definition. We have included only a subset of facets that are in most common use.

2.1. Basic Data Types

The knowledge model recognizes the following basic data types:

- integers
- floating point numbers
- double-precision floating point numbers
- strings
- boolean
- name of class

The original OKBC knowledge model supports (Lisp) symbols as one of the basic data types. We have dropped symbols from the list of supported data types because symbols are not commonly used in non-Lisp environments. OKBC-Lite also assumes that the domain of discourse includes the logical constants `true` and `false`.

2.2. Classes and Individuals

A *class* is a set of entities. Each of the entities in a class is said to be an *instance of* the class. An entity can be an instance of multiple classes, which are called its *types*. A class can be an instance of another class. A class which has instances that are themselves classes is called a *meta-class*.

Entities that are not classes are referred to as *individuals*. Thus, the domain of discourse consists of individuals and classes. Classes and individuals are generically referred to as entities.

The OKBC-Lite knowledge model assumes that every class and every individual has a unique identifier, also known as its *name*. The name may be a string or an integer, and is not necessarily human readable.

The class membership relation (called *instance-of*) that holds between an instance and a class is a binary relation that maps entities to classes. A class is considered to be a unary relation that is true for each instance of the class.

The relation *type-of* is defined as the inverse of relation `instance-of`. If A is an `instance-of` B, then B is a `type-of A`.

The *subclass-of* relation for classes is defined in terms of the relation `instance-of`, as follows. A class Csub is a subclass of class Csuper if and only if all instances of Csub are also instances of Csuper. The relation *superclass-of* is defined as the inverse of the relation `subclass-of`.

2.3. Own Slots and Own Facets

An entity has associated with it a set of *own slots*, and each own slot of an entity has associated with it a set of entities called *slot values*. For example, the assertion that Fred's favorite foods are potato chips and ice cream could be represented by the own slot `Favorite-Food` of the frame `Fred` having as values the frame `Potato-Chips` and the string "`ice cream"`.

An own slot of an entity has associated with it a set of *own facets*, and each own facet of a slot of a frame has associated with it a set of entities called *facet values*. For example, the assertion that the favorite foods of Fred must be edible foods could be represented by the facet `VALUE-TYPE` of the `Favorite-Food` slot of the `Fred` frame having the value `Edible-Food`.

2.4. Template Slots and Template Facets

A class has associated with it a collection of *template slots* that describe own slot values considered to hold for each instance of the class represented by the frame. The values of template slots are said to *inherit* to the subclasses and to the instances of a class. The values of a template slot are inherited to subclasses as values of the same template slot and to instances as values of the corresponding own slot. For example, the assertion that the gender of all female persons is female could be represented by template slot `Gender` of class frame `Female-Person` having the value `Female`. Then, if we created an instance of `Female-Person` called `Mary`, `Female` would be a value of the own slot `Gender` of `Mary`.

A template slot of a class has associated with it a collection of *template facets* that describe own facet values considered to hold for the corresponding own slot of each instance of the class represented by the class frame. As with the values of template slots, the values of template facets are said to inherit to the subclasses and instances of a class. Thus, the values of a template facet are inherited to subclasses as values of the same template facet and to instances as values of the corresponding own facet.

Note that template slot values and template facet values *necessarily* inherit from a class to its subclasses and instances. Default values and default inheritance are specified separately, as described in Section 2.6.

2.5. Collection Types for Slot and Facet Values

OKBC-Lite allows multiple values of a slot or facet to be interpreted as a collection type other than a set. The protocol recognizes three collection types: *set*, *bag*, and *list*. A bag is an unordered collection with possibly multiple occurrences of the same value in the collection. A list is an ordered bag. The knowledge model makes no commitment as to how values expressed in collection types other than `set` are combined during inheritance. Thus, OKBC-Lite guarantees that multiple slot and facet values of a frame stored as a bag or a list are retrievable as an equivalent bag or list *at that frame*. However, when the values are inherited to a subclass or instance, no guarantees are provided regarding the ordering of values for lists or the repeating of multiple occurrences of values for bags.

2.6. Default Values

The OKBC-Lite knowledge model includes a simple provision for default values for slots and facets. Template slots and template facets have a set of *default values* associated with them. Intuitively, these default values inherit to instances unless the inherited values are logically inconsistent with other assertions in the KB, the values have been removed at the instance, or the default values have been explicitly overridden by other default values. OKBC-Lite does not provide a means of explicitly overriding default values. Instead, OKBC-Lite leaves the inheritance of default values unspecified. That is, no requirements are imposed on the relationship between default values of template slots and facets and the values of the corresponding own slots and facets.

The default values on a template slot or template facet are simply available to the implementer to use in whatever way it chooses when determining the values of own slots and facets. OKBC-Lite guarantees that, unless the value of the `default` behavior is `none`, default values for a template slot or template facet asserted at a class frame will be retrievable *at that frame*. However, no guarantees are made as to how or whether the default values are inherited to a subclass or instance.

2.7. Knowledge Base

A knowledge base is a collection of classes, individuals, slots, slot values, facets, and facet values. A knowledge base is also known as a module.

2.8. Standard Classes, Facets, and Slots

The OKBC-Lite Knowledge Model includes a collection of classes, facets, and slots with specified names and semantics. It is not required that any of these standard classes, facets, or slots be represented in any given KB, but if they are, they must satisfy the semantics specified here.

The purpose of these standard names is to allow for canonical names for frequently used classes, facets, and slots. The canonical names are needed because an application cannot in general embed Literal references to frames in a KB and still be portable. This mechanism enables such Literal references to be used without compromising portability.

2.8.1. Classes

Whether or not the classes described in this section are actually present in a KB, OKBC-Lite guarantees that all these class names are valid values for the `VALUE-TYPE` facet described in Section 2.8.3.

`THING` *class*

`THING` is the root of the class hierarchy for a KB, meaning that `THING` is the superclass of every class in every KB.

`CLASS` *class*

`CLASS` is the class of all classes. That is, every entity that is a class is an instance of `CLASS`.

`INDIVIDUAL` *class*

`INDIVIDUAL` is the class of all entities that are not classes. That is, every entity that is not a class is an instance of `INDIVIDUAL`.

`NUMBER` *class*

`NUMBER` is the class of all numbers. OKBC-Lite makes no guarantees about the precision of numbers. If precision is an issue for an application, then the application is responsible for maintaining and validating the format of numerical values of slots and facets. `NUMBER` is a subclass of `INDIVIDUAL`.

`INTEGER` *class*

`INTEGER` is the class of all integers and is a subclass of `NUMBER`. As with numbers in general, OKBC-Lite makes no guarantees about the precision of integers.

`STRING` *class*

`STRING` is the class of all text strings. `STRING` is a subclass of `INDIVIDUAL`.

`SYMBOL` *class*

`SYMBOL` is the class of all symbols. `SYMBOL` is a subclass of `THING`.

`LIST` *class*

`LIST` is the class of all lists. `LIST` is a subclass of `INDIVIDUAL`.

2.8.2. Slots

`DOCUMENTATION` *slot*

`DOCUMENTATION` is a slot whose values at a frame are text strings providing documentation for that frame. Note that the documentation describing a class would be values of the *own slot* `DOCUMENTATION` on the class. The only requirement on the `DOCUMENTATION` slot is that its values be strings. That is,

```
   (=> (DOCUMENTATION ?F ?S) (STRING ?S))
```

2.8.3. Facets

The set of facets listed here is a strict subset of the set of the facets supported by OKBC-Lite.

`VALUE-TYPE` *facet*

The `VALUE-TYPE` facet specifies a type restriction on the values of a slot of a frame. Each value of the `VALUE-TYPE` facet denotes a class. A value C for facet `VALUE-TYPE` of slot S of frame F means that every value of slot S of frame F must be an instance of the class C.

A value for `VALUE-TYPE` can be a term of the following form:

```
   <value-type-expr> ::= (union <OKBC-class>*) | (set-of <OKBC-
value>*) |
                                 OKBC-class
```

A `OKBC-class` is any entity `X` for which `(class X)` is `true` or that is a standard OKBC class described in Section 2.8.1. A `OKBC-value` is any entity. The `union` expression allows the specification of a disjunction of classes (e.g., either a dog or a cat), and the `set-of` expression allows the specification of an explicitly enumerated set of possible values for the slot (e.g., either Clyde, Fred, or Robert).

`INVERSE` *facet*

The `INVERSE` facet of a slot of a frame specifies inverses for that slot for the values of the slot of the frame. Each value of this facet is a slot. A value S2 for facet `INVERSE` of slot S1 of frame F means that if V is a value of S1 of F, then F is a value of S2 of V.

`CARDINALITY` *facet*

The `CARDINALITY` facet specifies the exact number of values that may be asserted for a slot on a frame. The value of this facet must be a nonnegative integer. A value N for facet `CARDINALITY` on slot S on frame F means that slot S on frame F has N values.

For example, one could represent the assertion that Fred has exactly four brothers by asserting 4 as the value of the `CARDINALITY` own facet of the `Brother` own slot of frame `Fred`. Note that all the values for slot S of frame F need not be known in the KB. That is, a KB could use the `CARDINALITY` facet to specify that Fred has four brothers without knowing who the brothers are and therefore without providing values for Fred's `Brother` slot.

Also, note that a value for `CARDINALITY` as a template facet of a template slot of a class only constrains the maximum number of values of that template slot of that class, since the corresponding own slot of each instance of the class may inherit values from multiple classes and have locally asserted values.

`MAXIMUM-CARDINALITY` *facet*

The `MAXIMUM-CARDINALITY` facet specifies the maximum number of values that may be asserted for a slot of a frame. Each value of this facet must be a nonnegative integer. A value N for facet `MAXIMUM-CARDINALITY` of slot S of frame F means that slot S of frame F can have at most N values.

Note that if facet `MAXIMUM-CARDINALITY` of a slot S of a frame F has multiple values $N_1,...,N_K$, then S in F can have at most $\min(N_1,\ldots,N_K)$ values. Also, it is appropriate for a value for `MAXIMUM-CARDINALITY` as a template facet of a template slot of a class to constrain the number of values of that template slot of that class as well as the number of values of the corresponding own slot of each instance of that class since an excess of values for a template slot of a class will cause an excess of values for the corresponding own slot of each instance of the class.

`MINIMUM-CARDINALITY` *facet*

The `MINIMUM-CARDINALITY` facet specifies the minimum number of values that may be asserted for a slot of a frame. Each value of this facet must be a nonnegative integer. A value N for facet `MINIMUM-CARDINALITY` of slot S of frame F means that slot S of frame F has at least N values.

Note that if facet `MINIMUM-CARDINALITY` of a slot S of a frame F has multiple values $N_1,...,N_K$, then S of F has at least $\max(N_1,\ldots,N_K)$ values. Also, as is the case with the `CARDINALITY` facet, all the values for slot S of frame F do not need to be known in the KB.

Note that a value for `MINIMUM-CARDINALITY` as a template facet of a template slot of a class does not constrain the number of values of that template slot of that class, since the corresponding own slot of each instance of the class may inherit values from multiple classes and have locally asserted values. Instead, the value for the template facet `MINIMUM-CARDINALITY` constrains only the number of values of the corresponding own slot of each instance of that class, as specified by the axiom.

`NUMERIC-MINIMUM` *facet*

The `NUMERIC-MINIMUM` facet specifies a lower bound on the values of a slot whose values are numbers. Each value of the `NUMERIC-MINIMUM` facet is a number.

`NUMERIC-MAXIMUM` *facet*

The `NUMERIC-MAXIMUM` facet specifies an upper bound on the values of a slot whose values are numbers. Each value of this facet is a number.

`COLLECTION-TYPE` *facet*

The `COLLECTION-TYPE` facet specifies whether multiple values of a slot are to be treated as a set, list, or bag. No axiomatization is provided for treating multiple values as lists or bags because of the lack of a suitable formal interpretation for the ordering of values in lists of values that result from multiple inheritance and the multiple occurrence of values in bags that result from multiple inheritance.
The protocol itself makes no commitment as to how values expressed in collection types other than `set` are combined during inheritance. Thus, OKBC-Lite guarantees that multiple slot and facet values stored at a frame as a bag or a list are retrievable as an equivalent bag or list *at that frame*. However, when the values are inherited to a subclass or instance, no guarantees are provided regarding the ordering of values for lists or the repeating of multiple occurrences of values for bags.

2.8.4. Slots on Slots

The slots described in this section can be associated with slots. In general, these slots describe properties of a slot which hold at any frame that can have a value for the slot.

`DOMAIN` *slot*

`DOMAIN` specifies the domain of the binary relation represented by a slot frame. Each value of the slot `DOMAIN` denotes a class. A slot frame S having a value C for own slot `DOMAIN` means that every frame that has a value for own slot S must be an instance of C, and every frame that has a value for template slot S must be C or a subclass of C.

If a slot frame S has a value C for own slot `DOMAIN` and I is an instance of C, then I is said to be *in the domain of* S.
A value for slot `DOMAIN` can be a KIF expression of the following form:

```
   <domain-expr> ::= (union <OKBC-class>*) | OKBC-class
```
A `OKBC-class` is any entity `X` for which `(class X)` is `true` or that is a standard OKBC class described in Section 2.8.1.

Note that if slot `DOMAIN` of a slot frame S has multiple values $C_1,\ldots,C_N$, then the domain of slot S is constrained to be the intersection of classes C1,…, C_N. Every slot is considered to have `THING` as a value of its `DOMAIN` slot.

`SLOT-VALUE-TYPE` *slot*

`SLOT-VALUE-TYPE` specifies the classes of which values of a slot must be an instance (i.e., the range of the binary relation represented by a slot). Each value of the slot `SLOT-VALUE-TYPE` denotes a class. A slot frame S having a value V for own slot `SLOT-VALUE-TYPE` means that the own facet `VALUE-TYPE` has value V for slot S of any frame that is in the domain of S.

As is the case for the `VALUE-TYPE` facet, the value for the `SLOT-VALUE-TYPE` slot can be a KIF expression of the following form:

```
   <value-type-expr> ::= (union <OKBC-class>*) | (set-of <OKBC-
value>*) |
                              OKBC-class
```

A `OKBC-class` is any entity `X` for which `(class X)` is `true` or that is a standard OKBC class described in Section 2.8.1. A `OKBC-value` is any entity. The `union` expression allows the specification of a disjunction of classes (e.g., either a dog or a cat), and the `set-of` expression allows the specification of an explicitly enumerated set of values (e.g., either Clyde, Fred, or Robert).

`SLOT-INVERSE` *slot*

`SLOT-INVERSE` specifies the inverse relation for a slot. Each value of `SLOT-INVERSE` is a slot. A slot frame S having a value V for own slot `SLOT-INVERSE` means that own facet `INVERSE` has value V for slot S of any frame that is in the domain of S. Put another way, if V is the value of slot S1 of frame F, and S2 is the value of facet INVERSE of slot S1, then F must be the value of slot S2 from frame V. Two slots are inverses of one another if they encode the opposite relationship of one another, such as the relationships Parent and Child.

`SLOT-CARDINALITY` *slot*

`SLOT-CARDINALITY` specifies the exact number of values that may be asserted for a slot for entities in the slot's domain. The value of slot `SLOT-CARDINALITY` is a nonnegative integer. A slot frame S having a value V for own slot `SLOT-CARDINALITY` means that own facet `CARDINALITY` has value V for slot S of any frame that is in the domain of S.

`SLOT-MAXIMUM-CARDINALITY` *slot*

`SLOT-MAXIMUM-CARDINALITY` specifies the maximum number of values that may be asserted for a slot for entities in the slot's domain. The value of slot `SLOT-MAXIMUM-CARDINALITY` is a nonnegative integer. A slot frame S having a value V for own slot `SLOT-MAXIMUM-CARDINALITY` means that own facet `MAXIMUM-CARDINALITY` has value V for slot S of any frame that is in the domain of S.

`SLOT-MINIMUM-CARDINALITY` *slot*

`SLOT-MINIMUM-CARDINALITY` specifies the minimum number of values for a slot for entities in the slot's domain. The value of slot `SLOT-MINIMUM-CARDINALITY` is a nonnegative integer. A slot frame S having a value V for own slot `SLOT-MINIMUM-CARDINALITY` means that own facet `MINIMUM-CARDINALITY` has value V for slot S of any frame that is in the domain of S.

`SLOT-NUMERIC-MINIMUM` *slot*

`SLOT-NUMERIC-MINIMUM` specifies a lower bound on the values of a slot for entities in the slot's domain. Each value of slot `SLOT-NUMERIC-MINIMUM` is a number. A slot frame S having a value V for own slot `SLOT-NUMERIC-MINIMUM` means that own facet `NUMERIC-MINIMUM` has value V for slot S of any frame that is in the domain of S.

`SLOT-NUMERIC-MAXIMUM` *slot*

`SLOT-NUMERIC-MAXIMUM` specifies an upper bound on the values of a slot for entities in the slot's domain. Each value of slot `SLOT-NUMERIC-MAXIMUM` is a number. A slot frame S having a value V for own slot `SLOT-NUMERIC-MAXIMUM` means that own facet `NUMERIC-MAXIMUM` has value V for slot S of any frame that is in the domain of S.

`SLOT-COLLECTION-TYPE` *slot*

`SLOT-COLLECTION-TYPE` specifies whether multiple values of a slot are to be treated as a set, list, or bag. Slot `SLOT-COLLECTION-TYPE` has one value, which is either `set`, `list`, or `bag`. A slot frame S having a value V for own slot `SLOT-COLLECTION-TYPE` means that own facet `COLLECTION-TYPE` has value V for slot S of any frame that is in the domain of S.

3. XOL Syntax and Semantics

This chapter describes how to use XOL to define ontologies or, put another way, it describes how to build an XML document that conforms with the XOL DTD (Document Type Definition) in Appendix 1. It explains the semantics underlying each of the elements and attributes that are part of it.

The general form of an XOL document is as follows. The document begins with a **module** element, which identifies the single ontology contained in that XOL file. Next come a number of **class** elements, which define classes within that ontology. A series of **slot** elements lists the slots that are defined on those classes. Then a series of **individual** elements defines the objects within the ontology.

```
<module>

  <class>
  ...
  </class>

  <class>
  ...
  </class>
  ...

  <slot>
  ...
  </slot>

  <slot>
  ...
  </slot>
  ...

  <individual>
  ...
  </individual>
```

```
    <individual>
    ...
    </individual>
    ...

  </module>
```

3.1. The Module Section

An example module definition is

```
<module>
  <name>genealogy</name>
  <version>1.2</version>
  <documentation>An ontology for describing family
relationships.</documentation>
  <kb-type>ocelot-kb</kb-type>
  <package>user</package>
</module>
```

The first element of the document can be one of the five following names:

- **module**
- **ontology**
- **kb**
- **database**
- **dataset**

These element names are synonymous. XOL intentionally provides several synonyms for use within its definitions because different communities are used to using different terms for what are essentially the same underlying computer-science concepts.

The entire ontology description is given between the <**module**> and **</module>** tags. The **<name>** element within the module is required; it specifies the name of the ontology. The remaining module elements are optional (but if provided, they must be inserted in this position, and in this order):

kb-type or **db-type**

Specifies the database management system or knowledge-base management system in which this ontology was originally constructed, if any. Examples: **<kb-type>ocelot-kb</kb-type>** or **<kb-type>oracle</kb-type>**.

package

An optional element intended for LISP-based knowledge representation systems. It specifies the package in which all the symbols used in the knowledge base will be defined by default.

version

Specifies a version identifier for this ontology.

documentation

Provides documentation about the ontology.

3.2. The Class Section

Two example XOL class definitions are

```
<class>
<name>person</name>
<documentation>The class of all
people.</documentation>
</class>


<class>
<name>man</name>
<documentation>The class of all male persons.
</documentation>
<subclass-of>person</subclass-of>
</class>
```

The element **name** is required and is the name of the class. The element **documentation** is optional; it provides documentation about the class. Then follow elements of three possible types:

- **subclass-of**
- **instance-of**
- **slot-values**

The elements **subclass-of** and **instance-of** simply contain the name of the parent class of which the class is respectively a subclass or instance. The parent class is referred to by name, and that class must have been defined earlier in the document.

3.3. The Slot Section

Each definition in the slot section specifies a single slot that defines attributes of a class, or a relationship between classes. Consider the following example:

```
<slot>
  <name>year-of-birth</name>
  <documentation>An integer that represents the
year the person was born.</documentation>
  <domain>person</domain>
  <slot-cardinality>1</slot-cardinality>
  <slot-numeric-min>1800</slot-numeric-min>
  <slot-value-type>integer</slot-value-type>
</slot>

<slot>
  <name>brothers</name>
  <documentation>The brothers of a
person.</documentation>
  <domain>person</domain>
  <slot-value-type>man</slot-value-type>
</slot>

<slot>
  <name>citizenship</name>
  <documentation>Describes the citizenship status
of a person.</documentation>
  <domain>person</domain>
  <slot-value-type>(set-of citizen resident-alien
permanent-resident)</slot-value-type>
</slot>
```

The first XOL expression defines a slot called **year-of-birth** that applies to the class **person**. **Year-of-birth** is an integer-valued slot that can take at most one value, and that value is constrained to be greater than 1800. The second expression defines a slot called **brothers** whose values must be the names of individuals that are instances of the **man** class (see the example in Section 3.2). The third expression defines a slot called **citizenship** whose values must be taken from the specified list of three keywords.

The **name** and **documentation** elements follow the same usage as for classes.

The legal elements within a slot definition are described in detail in Section 2.8.4. In brief, those elements are

- **slot-value-type –** The datatype of the slot, which can be either the name of a primitive XOL datatype (such as **integer**), the name of a class defined in an ontology (such as **person** in this sample ontology), a set of keywords defined using the set-of constructor, or a combination thereof.
- **slot-inverse –** The slot that encodes the inverse semantic relationship of this slot. For example, parent and child encode inverse relationships.
- **slot-maximum-cardinality –** A lower bound on the number of values the slot may have.
- **slot-minimum-cardinality –** An upper bound on the number of values the slot may have.
- **slot-cardinality –** An exact number of values that the slot may have.
- **slot-numeric-minimum –** A lower bound on the value of a slot whose values are integers or real numbers.
- **slot-numeric-maximum –** An upper bound on the value of a slot whose values are integers or real numbers.
- **slot-collection-type –** For slots that can have more than one value, this element specifies whether those values are interpreted as a set, a list, or a bag.

3.4. The Individual Section

The **individual** element follows the same syntax as the class element except that there cannot be subclass-of elements inside the **individual** element. Example:

```
<individual>
  <name>fred</name>
```

```
    <documentation>The person Fred.
    </documentation>
    <instance-of>man</instance-of>
    <slot-values>
      <name>year-of-birth</name>
      <value>1960</value>
    </slot-values>
  </individual>
```

The **slot-values** element specifies the values of a slot within a given individual. And, in fact, **slot-values** elements may be used within classes as well to define slot values within a class. The **slot-values** element follows a more complex structure than preceding elements.

The element **name** corresponds to the name of the slot whose value is being specified (which must be one of the **slot** elements defined in the rest of the document). The one or more **value** elements specify the one or more values of that slot. The XOL definition for the individual Frank shows how to specify multiple slot values; Frank has two brothers, Fred and John.

```
  <individual>
    <name>frank</name>
    <documentation>The person Frank.
    </documentation>
    <instance-of>man</instance-of>
    <slot-values>
      <name>year-of-birth></name>
      <value>1962</value>
    </slot-values>
    <slot-values>
      <name>brothers></name>
      <value>fred</value>
      <value>john</value>
    </slot-values>
    <slot-values>
      <name>citizenship</name>
      <value>citizen</value>
    </slot-values>
  </individual>
```

3.5. Example XOL Ontology

The following genealogy ontology illustrates XOL in more detail. This ontology was constructed for purposes of illustration only and does not address many intricacies of genealogy and gender.

```
<?xml version="1.0" ?>
<!DOCTYPE module SYSTEM "module.dtd">

<module>
  <name>genealogy</name>
  <kb-type>ocelot-kb</kb-type>
  <package>user</package>

  <class>
    <name>person</name>
    <documentation>The class of all
persons.</documentation>
  </class>

  <class>
    <name>man</name>
    <documentation>The class of all persons who
define their gender as male.</documentation>
    <subclass-of>person</subclass-of>
  </class>

  <class>
    <name>woman</name>
    <documentation>The class of all persons who
define their gender as male.</documentation>
    <subclass-of>person</subclass-of>
  </class>
  <slot>
    <name>year-of-birth</name>
    <documentation>An integer that represents the
year the person was born.
    </documentation>
    <domain>person</domain>
    <slot-cardinality>1</slot-cardinality>
    <slot-numeric-min>1800</slot-numeric-min>
    <slot-value-type>integer</slot-value-type>
  </slot>

  <slot>
    <name>brothers</name>
    <documentation>The brothers of a person,
meaning the male siblings of a person who share a
parent with that person.</documentation>
    <domain>person</domain>
    <slot-value-type>man</slot-value-type>
  </slot>

  <slot>
    <name>citizenship</name>
    <documentation>Describes the current
citizenship status of a person with respect to
their primary country of
residence.</documentation>
    <domain>person</domain>
    <slot-value-type>(set-of citizen resident-
alien permanent-resident)</slot-value-type>
  </slot>

  <slot>
    <name>life-history</name>
    <documentation>A written history of the
person’s life.</documentation>
    <domain>person</domain>
    <slot-value-type>string</slot-value-type>
  </slot>

  <slot>
```

```
    <name>father-of</name>
    <documentation>father-of(X,Y) holds when X is
the father of Y.</documentation>
    <domain>man</domain>
    <slot-value-type>person</slot-value-type>
    <slot-inverse>has-father</slot-inverse>
  </slot>

  <slot>
    <name>has-father</name>
    <documentation>Has-father(X,Y) holds when the
father of X is Y.  Has-father and father-of are
inverses, which is why their domain and slot-
value-type are reversed: all fathers are male, but
the child of a father may be of either gender.
</documentation>
    <domain>person</domain>
    <slot-value-type>man</slot-value-type>
    <slot-inverse>father-of</slot-inverse>
  </slot>

  <individual>
    <name>John</name>
    <instance-of>man</instance-of>
    <slot-values>
       <name>year-of-birth</name>
       <value>1987</value>
    </slot-values>
    <slot-values>
      <name>citizenship</name>
      <value>permanent-resident</value>
    </slot-values>
    <slot-values>
      <name>has-father</name>
      <value>Carl</value>
    </slot-values>
  </individual>

  <individual>
    <name>Carl</name>
    <instance-of>man</instance-of>
    <slot-values>
       <name>year-of-birth</name>
       <value>1961</value>
    </slot-values>
    <slot-values>
      <name>father-of</name>
      <value>John</value>
    </slot-values>
    <slot-values>
      <name>life-history</name>
      <value>Carl worked hard all his
life.</value>
    </slot-values>
  </individual>

</module>
```

Acknowledgments

XOL was inspired by a study of ontology languages performed by the BioOntology Core group whose members included Neil Abernethy, Frank Olken, Robert E. Kent, Matt DeJongh, Peter Tarczy-Hornoch, David Benton, Dhiraj Pathak, Gregg Helt, Suzanna Lewis, Anthony Kosky, Eric Neumann, Dan Hodnett, Luca Toldo, and Thodoros Topaloglou. We thank Luca Toldo and Robin McEntire for extensive comments on this XOL specification document.

Appendix 1: XOL Document Type Definition

A document type definition (DTD) is a device of the XML language that is used to define the structure of a class of XML documents. A DTD defines the elements that are allowed within a class of XML documents, and the allowed ordering and nesting of those elements. The XOL DTD presented here defines the structure of XOL documents.

DTDs have two uses. They allow the authors of the XOL specification to communicate the legal structure of XOL documents to readers of that specification. In addition, DTDs can be used by XML parsing software to verify the validity of an XML document. XML documents can contain at their start the name of the DTD file to which that XML document must conform.

All XOL documents must be valid XML documents, but not all XML documents are valid XOL documents – only XML documents that follow the XOL DTD are legal XOL documents.

However, the DTD formalism cannot express all the rules necessary to define valid XOL documents. Additional constraints on the form of XOL documents are given in Appendix 2.

The XOL DTD is as follows:

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

<!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?, ( type
| slot-values )* >

<!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-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)>
```

Appendix 2: XOL Encoding Rules: Additional Rules Governing XOL Documents

XML DTDs to not have sufficient power to express all the necessary constraints on the form of XOL documents. Therefore, this appendix provides additional rules that XOL documents must follow.

1. The identifier provided in every **name** element within all **class**, **individual,** and **slot** elements must be unique within an XOL file. For example, the same name may not be used for two individuals, or for a slot and a class, within the same XOL file.
2. Each class must be defined earlier in an XOL file than are its subclasses.
3. Each class must be defined earlier in an XOL file than are its instances.
4. The identifier provided within the **subclass-of** and **instance-of** elements must be identical to the identifier within the **name** element of a class that is defined in that XOL file.
5. Only the subclass-of and instance-of elements for direct relationships to a parent class must be included in XOL files. Indirect relationships should not be included (e.g., if class A is a subclass of class B, which in turn is a subclass of class C, only the subclass-of link between A and B should be included in the XOL file). In addition, the superclass-of and type-of links that are the inverses of the subclass-of and instance-of links are optional.
6. The identifier provided within the **name** element of a **slot-values** element must be identical to the identifier within the **name** element of a slot that is defined in that XOL file.
7. Slots may be used only in classes and instances within their domain.
8. Values of a slot must obey the value-type definition for the slot.
9. Each class must be defined earlier in an XOL file than are its slots.

Bibliography

[CF97]

Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. *Open Knowledge Base Connectivity 2.0*. Technical Report KSL-98-06, Available from Knowledge Systems Laboratory, Stanford, July 1997. See also http://www.ai.sri.com/~okbc/

[CF98]

Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. "OKBC: A Foundation for Knowledge Base Interoperability." In *Proceedings of the National Conference on Artificial Intelligence,* pages 600-607, July 1998.

[GQ99]
Ian S. Graham and Liam Quin. *XML Specification Guide.* John Wiley and Sons, 1999.

[Gr93]
Thomas R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition* 5(2):199-220, 1993.

[Gu95]
N. Guarino and P. Giaretta, "Ontologies and Knowledge Bases: Towards a Terminological Clarification," in *Towards Very Large Knowledge Bases*, N.J.I. Mars, ed., pp25-32, IOS Press, Amsterdam, 1995.

[KC99]
Peter D. Karp, Vinay K. Chaudhri, and Suzanne M. Paley, "A Collaborative Environment for Authoring Large Knowledge Bases," *in Journal of Intelligent Information Systems,*" in press.