Handling XML with Florid3.0/FloXML

Addendum to Manual

Wolfgang May
may@informatik.uni-freiburg.de

November 2000

Abstract

This note describes the handling of XML in FloXML, which is the XML-enabled successor of the Florid system.

1 Introduction

Since version 2.0, Florid provides Web access functionality tailored to HTML sources (cf. [ML00], and the Mondial case study in [May99a]). Florid3.0, now called FloXML, extends this functionality with special XML handling features. Although the SGML functionality can directly be applied to XML sources, resulting in the F-Logic parse-tree model as described in [ML00], it does not really cover the *semantics* of XML sources. In this paper, we first investigate the specialties of *conceptual modeling* of XML documents and XML databases, focussing on the database aspect. Then, we describe the functionality implemented in Florid. The main advantages of our approach are the following:

- the object-oriented F-Logic data model:
  - a class hierarchy with nonmonotonic inheritance,
  - navigation along references,
  - schema reasoning,
- programming with Florid:
  - adding derived knowledge to the original sources,
  - integration of several sources.
  - implementation and evaluation of additional concepts, e.g., XML Schema, XPointer/XLinking in a declarative rapid-prototyping style.

2 XML

2.1 HTML, SGML, and XML

The intention of these markup languages is different – HTML means optical markup which incidentally contains also structural markup, SGML means structural and some logical markup, whereas XML provides logical markup and semantic structuring of a document which *can* be transformed into different structural and optical markup for presentations. Thus, for HTML and most SGML sources, the parse-tree is the natural representation. For XML sources, we have to distinguish between the *document* point of view and the *database* point of view.

2.2 The Core XML Data Model

It is assumed that the reader is familiar with the XML syntax. A complete XML document comes with a DTD which describes its structure. Here,

<!ELEMENT *elem_name* (*struct_spec*)>

describes the structure of the element. Elements can be empty, contain plain data, or can be constructed from subelements according to *struct_spec* (given in a regular-expression style). Additionally, elements can have attributes specified by

$$\begin{array}{lllll} \texttt{<!ATTLIST}\ \mathit{elem\_name} & \mathit{attr\_name}_1 & \mathit{attr\_type}_1 & \mathit{attr\_constr}_1 \\ & \vdots & \vdots & \vdots \\ & \mathit{attr\_name}_n & \mathit{attr\_type}_n & \mathit{attr\_constr}_n\texttt{>} \end{array}$$

Here, $attr\_type_i$ – roughly – distinguishes between value attributes and reference attributes, declares `id` attributes, and specifies if an attribute is scalar or multi-valued. Referencing is done via `ID` and `IDREF` attributes. $attr\_constr_i$ specifies if an attribute is mandatory or optional, also a simple kind of monotonic inheritance can be specified.

Example 1 (Mondial Database: DTD) *The structure of the* MONDIAL *XML (Version 2.0) database is defined by the following DTD:*

```
<!-- XML DTD "mondial-2.0.dtd":
  (Wolfgang May, may@informatik.uni-freiburg.de, March 2000)
  a hierarchical DTD for the MONDIAL database,
  containing e.g.,
  - scalar reference attributes (city/capital)
  - multivalued reference attributes (organization/member/country)
  - cross-references in both directions (organization/member/country,
    country/memberships)
  - a "boolean"/flag attribute: city/is_country_cap
  - reference attributes with more than one target class
    (river/to, references rivers, lakes, and seas) -->

<!ELEMENT mondial (country*,continent*,organization*,
                   mountain*,(sea*,river*,lake*,desert*,island*)*)>
<!ELEMENT country (name,area,population,
                   population_growth?,infant_mortality?,
                   gdp_total?,gdp_agri?,gdp_ind?,gdp_serv?,
                   inflation?,indep_date?,government?,encompassed+,
                   ethnicgroups*,religions*,languages*,border*,
                   province*,city*)>
<!ATTLIST country car_code ID #IMPLIED
                  capital IDREF #IMPLIED
                  memberships IDREFS #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT population (#PCDATA)>
<!-- note that population is also a subelement of city -->
<!ATTLIST population year CDATA #IMPLIED>
<!ELEMENT population_growth (#PCDATA)>
<!ELEMENT infant_mortality (#PCDATA)>
<!ELEMENT gdp_total (#PCDATA)>
<!ELEMENT gdp_ind (#PCDATA)>
<!ELEMENT gdp_agri (#PCDATA)>
<!ELEMENT gdp_serv (#PCDATA)>
<!ELEMENT inflation (#PCDATA)>
<!ELEMENT indep_date (#PCDATA)>
<!ELEMENT government (#PCDATA)>
<!ELEMENT encompassed EMPTY>
<!ATTLIST encompassed continent IDREF #REQUIRED
```

```
                      percentage CDATA #REQUIRED>
<!ELEMENT ethnicgroups (#PCDATA)>
<!ATTLIST ethnicgroups percentage CDATA #REQUIRED>
<!ELEMENT religions (#PCDATA)>
<!ATTLIST religions percentage CDATA #REQUIRED>
<!ELEMENT languages (#PCDATA)>
<!ATTLIST languages percentage CDATA #REQUIRED>
<!ELEMENT border EMPTY>
<!ATTLIST border country IDREF #REQUIRED
                 length CDATA #REQUIRED
                 justice IDREF 'org-UN'>

<!ELEMENT province (name,area?,population,city*)>
<!ATTLIST province id ID #REQUIRED
                   country IDREF #REQUIRED
                   capital IDREF #IMPLIED>

<!ELEMENT city (name,longitude?,latitude?,
                population*,located_at*)>
<!ATTLIST city id ID #REQUIRED
               is_country_cap CDATA #IMPLIED
               is_state_cap CDATA #IMPLIED
               country IDREF #REQUIRED
               province IDREF #IMPLIED>
<!ELEMENT longitude (#PCDATA)>
<!ELEMENT latitude (#PCDATA)>
<!ELEMENT located_at EMPTY>
<!ATTLIST located_at watertype (river|sea|lake) #REQUIRED
                     river IDREFS #IMPLIED
                     sea IDREFS #IMPLIED
                     lake IDREFS #IMPLIED>

<!ELEMENT organization (name,abbrev,established?,members*)>
<!ATTLIST organization id ID #REQUIRED
                       seat IDREF #IMPLIED>
<!ELEMENT abbrev (#PCDATA)>
<!ELEMENT established (#PCDATA)>
<!ELEMENT members EMPTY>
<!ATTLIST members type CDATA #REQUIRED
                 country IDREFS #REQUIRED>

<!ELEMENT continent (name,area)>
<!ATTLIST continent id ID #REQUIRED>

<!ELEMENT river (length?,name,to*,located*)>
<!ATTLIST river id ID #REQUIRED
                country IDREFS #REQUIRED>

<!ELEMENT length (#PCDATA)>
<!ELEMENT to EMPTY>
<!ATTLIST to watertype (river|sea|lake) #REQUIRED
             water IDREF #REQUIRED>

<!ELEMENT located EMPTY>
<!ATTLIST located country IDREF #REQUIRED
                  province IDREFS #IMPLIED>

<!ELEMENT lake (name,area?,depth?,located*)>
```

```
<!ATTLIST lake id ID #REQUIRED
               country IDREFS #REQUIRED>
<!ELEMENT depth (#PCDATA)>

<!ELEMENT sea (name,area?,depth?,located*)>
<!ATTLIST sea id ID #REQUIRED
              country IDREFS #REQUIRED
              bordering IDREFS #REQUIRED>

<!ELEMENT desert (name,area?,located*)>
<!ATTLIST desert id ID #REQUIRED
                 country IDREFS #REQUIRED
                 climate NMTOKENS #FIXED 'dry aride'
                 temperature NMTOKEN 'hot'
                 ground (sand|boulders|rocks|snow) 'sand'>

<!ELEMENT island (name,area?,longitude?,latitude?,located*)>
<!ATTLIST island id ID #REQUIRED
                 country IDREFS #REQUIRED>

<!ELEMENT mountain (name,longitude?,latitude?,height?,located*)>
<!ATTLIST mountain id ID #REQUIRED
                   country IDREFS #REQUIRED>
<!ELEMENT height (#PCDATA)>
```

Example 2 (Mondial Database: the XML instance) *The following is an excerpt of an XML representation of the* MONDIAL *database [May99a] (Version 2.0): The full database can be found at [Mon].*

```
<continent id="europe">
  <name>Europe</name>
  <area>9562488</area>
</continent>

<country car_code="D"
  capital="cty-Germany-Berlin"
  memberships="... org-CERN org-ESA org-EU org-IOC org-NATO org-UN org-WHO ...">
  <name>Germany</name>
  <population>83536115</population>
  <encompassed continent="europe" percentage="100"/>
  <ethnicgroups percentage="95.1">German</ethnicgroups>
  <ethnicgroups percentage="2.3">Turkish</ethnicgroups>
  <ethnicgroups percentage="0.4">Greeks</ethnicgroups>
  <ethnicgroups percentage="0.7">Italians</ethnicgroups>
  ...
  <religions percentage="37">Roman Catholic</religions>
  <religions percentage="45">Protestant</religions>
  <languages percentage="100">German</languages>
  <border country="F" length="451"/>
  <border country="A" length="784"/>
  <border country="CZ" length="646"/>
  <border country="CH" length="334"/>
  <border country="PL" length="456"/>
  <border country="B" length="167"/>
  <border country="L" length="138"/>
  <border country="NL" length="577"/>
  <border country="DK" length="68"/>
  <province id="prov-cid-cia-Germany-2" capital="cty-cid-cia-Germany-9" country="D">
    <name>Baden Wurttemberg</name>
```

```
    <area>35742</area>
    <population>10272069</population>
    <city id="cty-cid-cia-Germany-9" country="D" province="prov-cid-cia-Germany-2">
      <name>Stuttgart</name>
      <longitude>9.1</longitude>
      <latitude>48.7</latitude>
      <population year="95">588482</population>
    </city>
    <city id="cty-cid-cia-Germany-24" country="D" province="prov-cid-cia-Germany-2">
      <name>Karlsruhe</name>
      <population year="95">277011</population>
    </city>
    <city id="cty-cid-cia-Germany-41" country="D" province="prov-cid-cia-Germany-2">
      <name>Freiburg im Breisgau</name>
      <population year="95">198496</population>
    </city>
    ...
  </province>
  <province id="prov-cid-cia-Germany-3" capital="cty-Germany-Munich" country="D">
    <name>Bayern</name>
    <area>70546</area>
    <population>11921944</population>
    <city id="cty-Germany-Munich" country="D" province="prov-cid-cia-Germany-3">
      <name>Munich</name>
      <longitude>11.5667</longitude>
      <latitude>48.15</latitude>
      <population year="95">1244676</population>
    </city>
    ...
  </province>
  ...
</country>

<organization id="org-EU" seat="cty-cid-cia-Belgium-2-0">
  <name>European Union</name>
  <abbrev>EU</abbrev>
  <established>07 02 1992</established>
  <members type="member"
           country="GR F E A D I B L NL DK SF S IRL P GB"/>
  <members type="membership applicant"
           country="AL CZ H SK LV LT PL BG RO EW M CY"/>
</organization>
```

3 Mapping XML Documents to F-Logic

3.1 Data Models on the Web

Depending on the "sophistication" of a Web site, there are several kinds of coexisting data models on the Web:

HTML pages: for most HTML pages, HTML is just a *presentation* of semi-structured data, but not a conceptual model. The conceptual model has (hopefully) been known to the author or is implemented in the underlying data source, but it is not accessible to the user. Thus, the application has to generate its own conceptual model if it wants to work on the data.

Sometimes, HTML structures (e.g., tables) are not only an auxiliary structuring facility, but contain structured information which is naturally presented as a table or a relation. There, table-rows (tuples), table-header (attributes) and table-data entries (attribute values) are the conceptual model of the information.

SGML documents: For providing documents, i.e., information which is inherently structured by sections, paragraphs, headers etc., the SGML parse-tree is not only an auxiliary format, but the natural conceptual model of the information.

XML: The hierarchical XML data model and the derived DOM model provide a "full" conceptual model for the underlying semistructured data which is very similar to the object-oriented model. Due to its flexibility it is used as a self-describing format for electronic data interchange.

For extracting data from the Web, a framework must be able to handle all of these models. Thus, a *unified* internal model is required which covers the above models and supports wrapping functionality to lift data from the "lower"-level model (HTML) to the "higher"-level model (XML, or an internal data model). Also, since both the sources themselves and the XML framework as a whole are subject to change (e.g., extensions like XML Schema, XPointer etc), the language should be extensible and should support rapid prototyping.

In principle, both general SGML and XML sources can be handled by using an arbitrary SGML parser for fetching the data into our system, resulting in a representation of the SGML/XML parse-tree. This is in fact a very poor kind of "data model" – especially when compared with the data model which is effectively provided by the XML data. Thus, a language smoothly integrating these data sources is needed: If possible, data is accessed as self-describing XML data with semantic tags, otherwise it is handled on the parse-tree level which is then complemented by wrapper functionality.

For coping with these requirements, we extended our existing framework for HTML Web access with support for XML features: F-Logic has been used for modeling semistructured data in general [LHL+98]. An integrated internal model for the Web structure and for HTML Web data has been presented in [May99b]. In [MHLL99], the use of generic wrappers for common tasks when handling HTML sources has been illustrated. In the present paper, we show how XML is seamlessly integrated into this framework.

3.2 Document versus Database Point of View

In fact, there are two coexisting basic conceptual models of XML data: parse-trees, and a semistructured *data model* based on application-specific notions independent from a classical document structure.

Documents. A document is based on a hierarchical structure, e.g., an introduction, some sections which contain subsections which again contain paragraphs, tables, figures etc., an appendix, a bibliography etc. The notions of structuring are roughly the same for all documents. The information contained in the document consists of its (textual) contents, and of the "meta information" given by its hierarchical structure, and the ordering of its contents. Without their context, most of the items (which can bee seen as subelements, parse-tree nodes etc) become useless. Access to the contents mostly is done by navigation in the hierarchical document structure and searching for objects which satisfy some condition. Queries start with some high-level object and navigate downwards through the document hierarchy. This idea mirrors e.g. in XPath's nested path expressions and filters.

Databases. A database describes (i) sets of objects, (ii) their properties, and (ii) their relationships. Most of these objects are still useful without the surrounding document.

The conceptual model can, e.g., be given by an ER model or an UML specification. The database itself implements a *data model*, e.g., the relational model, an object-relational model, or a fully object-oriented model [CB00]. For an application, it should not matter, which of the above data models has been chosen. Also, the *representation* does not matter, neither the *ordering* of the database contents (in general, these database models already abstract from the ordering).

When accessing the database, often several objects are accessed which have certain relationships. This idea mirrors in join constructs including several sets of candidate objects of SQL/OQL-like languages.

Querying Languages. The document point of view is represented by XPath-like languages which support navigation downwards a hierarchy, filtering by a predicate on each level. The database point of view is represented by SQL/OQL-like languages which allow to start a query with several (sets of) objects, joining them, following references etc.

Join concepts, or other means for navigating orthogonally to the underlying hierarchical structure are not fully supported by XPath (only semi-joins, no following of references in path expressions). On the other hand, database languages do not support typical hierarchy-navigation functionality such as searching for an arbitrarily deep subelement, or indexing or ordering of answers.

In our approach, we focus on the database point of view, without excluding document-style functionality in the model.

3.3 Subelements vs. Attributes.

XML provides two different kinds of properties (orthogonal to the concepts of literal-valued properties and object-valued properties/relationships known from object-oriented databases): subelements or attributes.

A popular strategy when deciding if something should be modeled by a subelement or by an attribute is

- everything which is "contents" is modeled as subelements (document view: the text)
- everything which is information *about* the context is modeled as attributes (e.g., markup)

This is a very practical solution in the document view: the contents is in fact contained as `PCDATA` (in the subelements) whereas everything else is markup (subelement nesting, tags) or meta-information (attribute values). For this, XPath provides the text() function for nodes which results in the (recursive) textual contents, "stripping" off the attributes.

For the database view, this strategy is not necessarily appropriate. Since a database has no "textual" contents, the choice between attributes and subelements is more a matter of taste. Additionally, it is a matter of taste if a relationship is represented by an element-subelement relationship, or by a reference attribute. The following slightly restricts the choice:

- attributes do not allow "complex" information, such as parameters (e.g., *berlin*:city[population@(1995)→3472009; coordinates→(13.3, 52.45)].
  here, complex attributes could be proposed:
  ```
  <!ATTLIST city   population pop_struct #IMPLIED
                   coordinates coord_struct #IMPLIED>
  <!STRUCT pop_struct   year NUMERIC
                        value NUMERIC >
  <!STRUCT coord_struct   longitude NUMERIC
                          latitude NUMERIC >
  <city name="Berlin"  population="1995; 3472009"
                       coordinates="13.3; 52.45"/>
  ```
- only attributes provide referencing functionality: simple relationships between objects can be represented via `IDREF` attributes. Here, the XML Schema proposal [XML99] provides additional expressiveness for specifying complex relationships.

Most concepts around XML make a distinction between subelements and attributes (e.g., XPath, XML-QL, DOM and SAX APIs) by the convention that subelements are addressed via their pure name (e.g., country/name), whereas attributes are addressed via their name prefixed with "@", e.g., country/@area. In our approach, we do *not* make this distinction, but address both just by their name: country.name. Together with a silent reference resolving by path expressions, our structures are highly independent from the actual XML-representation, providing full navigation along semantical references. Especially, for the same ontology, there can be numerous different XML representations:

- a flat model with crossreferences,
- different hierarchical models,
- models representing everything except references by subelements,
- models representing nearly everything by attributes.

One of the main ideas behind or approach can be summarized as follows:

> *For XML database applications, the semantics of a database is independent from actual XML representation, i.e., the ordering, and the representation by attributes or subelements, and from splitting over several interlinked files.*

3.4 Overview of Implemented Transformations

An important issue when accessing and interpreting documents is if the source should be regarded as a document with its parse-tree or as a database. As already shown, for XML sources there are again different possibilities how to represent the document in the internal model, depending on what is provided with the XML source, and what the user intends to do with the XML source(s). The F-Logic data model covers both the DTD or XML Schema (by signature atoms) and the XML instance; it even holds more explicit semantics since the target classes of reference attributes are known. Additionally, it allows to extend the data with a class hierarchy and inheritance. FLORID provides the following functionality:

- "document" view: the source is interpreted as a hierarchically structured parse-tree where the ordering of items is relevant and references merely are cross-references as known from documents.
- "database" view: the source is seen as an (object-oriented) database where objects and their properties are described. Here, references are seen as object-valued properties. The hierarchical structure also induces relationships between objects.
- automatical resolving of references inside the source (only possible when a DTD for distinguishing `ID` and `IDREF` attributes is available),
- if a DTD to the document is provided (an declared in the XML prolog), it can be used,
- accessing several documents using several contexts.

The following Web access methods which implement the features described above are provided by the current Web interface:

u**.get:** The ASCII representation of a Web page is fetched into the system.

u**.parse@(prolog):** only the prolog is evaluated, giving information about the documenttype, the DTD, and, in case it uses an external DTD, the location of the DTD.

u**.parse@(dtd, *context*):** u is parsed as a DTD, the schema information about a document type is extracted and mapped to an internal representation. If a *context* is given (`nil` otherwise), all names (i.e., class names and method names) are augmented with the context, in a similar way as XML namespaces (cf. Section 7).

u**.parse@(sgml, *references*, *context*):** the document accessible via u is regarded as an SGML document, i.e., the parse-tree is generated. If *references* is `true`, references are resolved automatically. If a *context* is given (`nil` otherwise), all names are augmented with the context.

u**.parse@(xml, *references*, *ordering*, *schema*, *context*):** the document accessible via u is regarded as an XML source from which an object-oriented representation is generated. If *references* is `true`, then references are resolved automatically. If *ordering* is `true`, the parse-tree is embedded into the object-oriented model. The *schema* parameter selects if scalar and multivalued properties should be distinguished by the DTD given in the XML source or by another DTD or XML Schema file (given by an url), or if a resident F-Logic signature should be used. If a *context* is given (`nil` otherwise), all atoms are augmented with the context.

Environment Variables. The above functionality is only available when the environment variables

SP_ENCODING=XML and
SGML_CATALOG_FILES=*Florid-Directory*/sgml/xml.soc

are set.

3.5 XML Prolog Information

XML documents contain some important meta information about the document in the prolog:

- an xml processing instruction of the form
  <?xml version="*version-spec*" encoding="*encoding-spec*" ?>
- a doctype declaration of the form
  <!DOCTYPE *doctype-name* SYSTEM *dtd-filename*> or
  <!DOCTYPE *doctype-name* PUBLIC *dtd-filename*>

The doctype of an XML document is made a property of the document. Also, the filenames of systemid or publicid then become attributes of the webdoc[1]:

```
url[parse@(xml,...)⇒xmldoc].
xmldoc :: webdoc.
webdoc[doctype⇒doctype;
        filled in if a SYSTEM resource is declared:
        systemurl⇒⇒url;        % the url of the DTD
        system⇒⇒string;        % filename of the DTD
        filled in if a PUBLIC resource is declared:
        publicurl⇒⇒url;        % the url of the DTD
        public⇒⇒string].       % filename of the DTD

myurl.parse@(xml,...):webdoc[
            doctype→mondial;
            systemurl→"http://www-server/dir/mondial-2.0.dtd";
            system→"mondial-2.0.dtd"].
```

The prolog information determines several details of handling the document to be parsed:

- Document type HTML requires several specialties of HTML parsing (e.g., empty tags),
- For the database view on XML documents, the DTD can be used for determining the target schema, e.g., if properties are scalar or multivalued, optional or required, literal or object-valued (cf. Section 5).

4 Document View: Parse-Tree Representation of SGML sources

When accessing SGML sources from the *document* point of view, the conceptual model to be used is simply the parse-tree model. The internal representation is based on parse-nodes which are naturally given by the source. Different kinds of parse-nodes are induced by the tags, e.g., sections, subsections, paragraphs, tables, lists etc.

Every tag <name> of the SGML document defines a class name which contains all elements of the SGML document with this tag; name – as a class – is a subclass of the class node, name – now as an object – becomes an element of the class tag. Instances of node have attributes and subelements (children). Children are generically represented according to the signature

node[child@(tag,int)⇒node]

[1] we decided not to make them properties of the doctype since different people can declare the same doctype (name) with different DTDs (and DTD urls).

where the tag position holds the subelement's tag, and the int position enumerates the children of each node. Attributes also define properties of their node:

node[attribute⇒string].

The possible values of tag and attribute are given by the DTD or by the document instance (hopefully, equivalent – if the document is valid).

Example 3 (SGML parse-tree) *We illustrate the schema data for a simple sectioning markup. Let*

$\mathit{section}_3$ = (<section id="sec-architecture" author="X.Y.">
header_3 para_{31} figure_{31} para_{32} . . . </section>) ,

header_3 = (<header> The Architecture </header>) ,

para_{3i} = (<paragraph> . . . </paragraph>) ,

figure_{31} = (<figure id="fig-architecture" src="pictures/architecture.ps"/>) .

Then, the following F-Logic representation of the parse-tree is generated:

section:tag. section::node. header:tag. header::node.
paragraph:tag. paragraph::node. figure:tag. figure::node.

$\mathit{section}_3$:section[author→"X.Y."; id→"sec-architecture";
child@(header,1)→header_3; child@(par,2)→para_{31};
child@(figure,3)→figure_{31}; child@(par,4)→para_{32}].

figure_{31}:figure[id="fig-architecture"; src="pictures/architecture.ps"].

We can also define

Element[Subel↠X] :- Element:node, Element[child@(Subel,N)→X]

yielding

$\mathit{section}_3$[header↠{header_3}]

for enabling navigation by tag names.

Here, it is a crucial feature of F-Logic that objects can simultaneously occur as objects, classes and methods (see, e.g., header above). This allows for powerful schema integration rules based on meta-reasoning.

In current SGML/XML, the data contents of attributes and children is always seen as a string (`CDATA`, `NMTOKEN(S)`, and `PCDATA`); there is no inherent concept of numerical values (the XML Schema proposal [XML99] supports numbers). In our approach, all textual contents which matches the expression `\A[0-9]*(\.[0-9]*)?\Z` are interpreted as numbers (silently casting numbers back into strings when a string value is expected).

Data Contents. SGML distinguishes between parse-nodes (i.e., <*tag*>. . . </*tag*>) and their `PCDATA` contents, thus, the first child of a `PCDATA` element yields its `PCDATA` contents (similar to XSL's text() function[2]); we give it the internal "tag" name pcdata:

header_3[child@(pcdata,1)→"The Architecture"].

The modeling straightforwardly extends to mixed contents:

Example 4 *For the mixed-contents paragraph*

para_{31} = (<paragraph > The architecture as shown in Figure
<crossref target="figure-architecture"/> <!– declared as `IDREF` –>
has been presented in
<cite href="http:..."/> <!– here also XPointer can be used –>
</paragraph >)

results in

[2] note that text() results in the concatenation of the text contents of all its children nodes; nevertheless, for pure `PCDATA` elements, this is the same; the same approach has been applied for XML-QL.

$para_{31}$[child@(pcdata,1)→ "The architecture as shown in Figure";
child@(crossref,2)→$crossref_{311}$;
child@(pcdata,3)→ "has been presented in";
child@(cite,4)→$cite_{311}$].

$crossref_{311}$[target→ "figure-architecture"].
$cite_{311}$[href→ "http:..."].

Optionally, references can be resolved. For the above example, this would result in

$crossref_{311}$[target→$figure_{31}$].

An extension when several documents are accessed which should be integrated is described in Section 7.

5 Database View: Object Model

When accessing well-designed XML sources, the intended conceptual model is in general not the parse-tree of a source. Instead, the XML source can directly be translated into an object-oriented model of the application domain. We first give an overview how the result will look like.

Here, we consider XML documents where either no schema information is given, or schema information is given by a DTD. In Section 8, an extended approach is described which handles XML Schema meta-information.

XML element types simultaneously define classes (collecting the elements) and methods (of the parent element, resulting in the appropriate subelements) which result in that class:

```
<!ELEMENT country (city*|...)>
<!ELEMENT city (...)>

<country name="Switzerland">
    <city id="city-ch-bern" name="Bern"/>
    <city id="city-ch-geneva" name="Geneva"/>
</country>
```

country[city⇒⇒city]. *% method city applied to objects of class*
% country results objects of class city

switzerland[city→→{*bern*,*geneva*,...}]. *bern*:city. *geneva*:city.

`CDATA`, `NMTOKEN(S)`, and `ID` attributes define properties with literal results (i.e., strings and numbers):

```
<!ELEMENT country (...)>
<!ATTLIST country  name CDATA #REQUIRED
                   area CDATA #IMPLIED
                   industry NMTOKENS #IMPLIED>

<country name="Switzerland" area="41290"
    industry="machinery ...">
```

country[name⇒string; area⇒numeric; industry⇒⇒string].
switzerland[name→ "Switzerland"; area→ "41290"; industry→→{ "machinery", ...}].
organization[abbrev⇒string]. *eu*[abbrev→ "EU"].

Reference attributes (`IDREF(S)`), define object-valued properties (relationships):

<!ELEMENT country (...)>
<!ATTLIST country capital IDREF #REQUIRED
memberships IDREFS #IMPLIED>

<!ELEMENT organization (...)>
<!ATTLIST organization abbrev ID #REQUIRED
seat IDREF #IMPLIED>

<country name="Switzerland" capital="city-bern" memberships="EU UN ..."/>
<organization abbrev="EU" seat="city-brussels"/>

country[capital⇒city; memberships⇒⇒organization]. org[seat⇒city].
switzerland[capital→*bern*; memberships→→{*eu*, *un*, . . . }]. *eu*[seat→*brussels*].

We come back to the details in the following sections.

5.1 SGML/XML "Schema" Information

From a DTD or XML Schema description, the schema knowledge can be extracted – describing properties of element *types*: Every element type e has a property contents (elements, pcdata, mixed, or empty), and every attribute a or subelement name s defines an object x such that e[attribute@(a)→x] or e[subelement@(s)→x] which has properties type (an elementtype for subelements, and literal, id, or ref, for attributes), card (scalar or multivalued), and minCard (0 (mandatory) or 1 (optional)) as described by the following F-Logic signature (closely related to the XML Schema notions):

```
elementtype[contents => contentstype;
            attribute =>> attribute;
            element =>> element].
empty:contentstype.
pcdata:contentstype.
elements:contentstype.
mixed:contentstype.

attribute[name => string; type => attrtype;
          card => cardinality; minCard => integer].
element[name => string; type => elementtype;
        card => cardinality; minCard => integer].
mvd:cardinality.
sc:cardinality.
id:attrtype.
token:attrtype.
cdata:attrtype.
ref:attrtype.
enumeration::attrtype.  %  source-dependent enum datatypes
enumval::object.        %  source-dependent enum values
fixed:inhType.
default:inhType.
attribute[inheritance => inhType;
          value => object;     %% in case of card->sc
          value =>> object].   %% in case of card->mvd
enumeration[name => string;
            values =>> enumval].
```

5.2 Data Transformation

The F-Logic data model does not distinguish between attributes and subelements – here, both are properties. Generally, scalar attributes and subelements (i.e., subelements which appear at most once inside some other element) define scalar methods whereas multivalued ones define multivalued methods. XML elements and attributes can be modeled as objects, or as values of a property of an object.

5.2.1 Elements

All tags (or, "element types") define classes. On the instance level, these tags define methods of their host elements/objects which result in objects of that class. Dependent on the (maximal) cardinality of a subelement, *scalar* and *multivalued* properties are distinguished.

```
<!ELEMENT country (map|city*|...)>
<!ELEMENT map (...)>
<!ELEMENT city (...)>

<country name="Switzerland">
    <map src="..."/>
    <city id="city-ch-bern" name="Bern"/>
    <city id="city-ch-geneva" name="Geneva"/>
</country>

country[city⇒⇒city; map⇒map].         % method city applied to objects of class
                                      % country results objects of class city
switzerland[map→map-ch; city↠{bern,geneva,...}].
bern:city.   geneva:city.
map-ch:map[src→...].
```

5.2.2 Literal Attributes

Attributes with literal semantics, i.e., CDATA, NMTOKEN(S), and ID attributes define properties with literal results. Since the DTD does not distinguish between strings and numbers, for a given DTD, only literal can be derived:

```
<!ELEMENT country (...)>
<!ATTLIST country   name CDATA #REQUIRED
                    area CDATA #IMPLIED
                    industry NMTOKENS #IMPLIED>

<!ELEMENT organization (...)>
<!ATTLIST organization   abbrev ID #REQUIRED
                         established CDATA #IMPLIED>

<!ELEMENT map (...)>
<!ATTLIST map   src CDATA #REQUIRED >

<country name="Switzerland" area="41290"
    industry="machinery ...">

<organization abbrev="EU" established="07 02 1992">

country[name⇒literal; area⇒literal; industry⇒⇒literal].
switzerland[name→"Switzerland"; area→"41290"; industry↠{"machinery", ...}].
organization[abbrev⇒literal; established⇒literal].
eu[abbrev→"EU"; established→"07 02 1992"].
map[src⇒literal].
```

Here, proposals like XML Schema [XML99] which distinguish literal data types provide much more explicit information: strings and numeric values can be distinguished; also subclasses of string can be used, e.g., date and urls. For the datatype hierarchy, the F-Logic concepts for expressing a class hierarchy can be used:

string::literal.   numeric::literal.
date::string.   url::string.
country[name⇒string; area⇒numeric; industry⇒⇒string].
organization[abbrev⇒string; established⇒date].
map[src⇒url].

Literal Values. In XML, all literal values are represented as strings, occurring as `CDATA`, `NMTOKEN`, or `ID` (in attributes) or `PCDATA` (in element contents), especially, numeric values are not supported. In the derived F-Logic model, we distinguish numeric values from strings. On the data level, every string representing a number is automatically converted to a number (for applying string operations, numbers are silently casted into strings).

5.2.3 Reference Attributes

For reference attributes (`IDREF, IDREFS`), we pay for the XML decision that references are taken from a global set of `ID`s without specifying which should be the type of the referenced element (e.g., the DTD does not tell that the capital must reference a city – referencing an organization would also be valid). Again, using an XML Schema specification, the appropriate classes can be derived. In our example, the signatures of the reference attributes are as follows, specifying the class of the referenced objects:

country[capital⇒city; memberships⇒⇒org].
org[seat⇒city].

5.2.4 Elements and Text Contents: "Annotated Literals"

The XML data model distinguishes between elements (i.e., nodes of the form <*tagname*> ... </*tagname*>) and their `PCDATA` contents. The handling of literal-valued attributes and actually literal-valued subelements differs:

For the first representation,

<country name="Germany"/>

//country/@name yields "Germany"

For the second representation,

<country>
  <name>Germany</name>
</country>

(i) //country/name yields <name>Germany</name> and
(ii) //country/name.text() yields "Germany".

"Living" in an XML model, there is a difference between (i) an element, and (ii) a string as a result. In contrast, in our approach, we use an object-and-literal-based model where we intentionally want to have literals as literals, independent of their representation as attribute values or subelements: the application-level semantics of country.name should always be the same: the string "Germany".

Thus, we want to treat `PCDATA` contents of a subelement as the *literal value* of a property. In presence of attributes, e.g.,

<city name="Berlin">
  <population year="1995"> 3472009 </population>
</city>

these attributes do not describe the value (here, the integer 3472009), but the relationship between the property population and the value.

Thus, in our translation, `PCDATA` elements with attributes behave as *annotated literals*, i.e., they are modeled as complex objects with a text property

city[population⇒population].
population[year⇒numeric; pcdata⇒numeric].
berlin[population→*pop-b-95*].
pop-b-95:population[year→1995; pcdata→3472009].

but act as their literal values when used in string/numeric comparisons, arithmetics, or (optionally) in the output. Thus, the user can treat country.population like a numerical value which is additionally annotated with a year attribute:

Example 5 (Annotated Literals) *The property city..population is an annotated literal:*

?-_:city[name→"Berlin"; population↠P].
  P/3472009
?-_:city[name→"Berlin"; population↠P[year→Y]].
  P/3472009   Y/1995
?-_:city[name→"Berlin"]..population[year→1995] > 3000000.
  true

Although, 3472009[year→1995] does not *hold.*

The "=" comparison predicate is interpreted as *object* equality; thus, it does *not* resolve annotated literals. Instead, the built-in equiv predicate evaluates to true, if, after resolving arguments, the values are equivalent:

?-_:city[name→"Berlin"; population↠P[year→Y]], P = 3472009.
  false
?-_:city[name→"Berlin"; population↠_[year→Y; pcdata↠P]], P = 3472009.
  true
?-_:city[name→"Berlin"; population↠P[year→Y]], equiv(P,3472009).
  true

In Florid, automatic resolving of annotated literals in the output can be turned on/off with a printer switch (default is "off"):

?- sys.prn.annotatedLiterals@("on").
?- _:city[name→"Berlin"; population↠P].
P/3472009
?- sys.prn.annotatedLiterals@("off").
?- _:city[name→"Berlin"; population↠P].
P/f0_1234

When using output redirection to a file with the intention to consult the file with Florid again, annotated literals should be turned off.

5.2.5 Enumerations

With DTDs and XML Schema, in addition to predefined types like `NMTOKEN`, `IDREF`, or `CDATA`, the range of an attribute can be specified by enumerating the possible values. Note that attributes with an enumerated type can only be scalar.

In our approach, we decided to represent enumerate types as classes, treating them the same way as reference attributes. This modeling directly corresponds to the XML Schema approach.

```
<!ELEMENT city (..., located_at*)>
<!ATTLIST city name ID #REQUIRED>
<!ELEMENT located_at EMPTY >
<!ATTLIST located_at  watertype (river, lake, sea) #REQUIRED
                      water IDREF #REQUIRED>
<!ELEMENT river (...)>
>!ATTLIST river name ID>

<river name="Donau"/>
<city name="Vienna">
    <located_at watertype="river" water="Donau"/>
</city>
```

becomes

city[name⇒string; located_at⇒⇒located_at].
located_at[watertype⇒*anon_type*; water⇒water].
river:*anon_type*.
lake:*anon_type*.
sea:*anon_type*.

vienna:city[name→"Vienna"; located_at→→{*vienna-loc*}].
vienna-loc:located_at[watertype→river; water→*donau*].
donau:river[name→"Donau"].

(in XML Schema, *anon_type* could be declared as a simpleType with the name watertype.)

5.2.6 Default Declarations

DTDs and XML Schema allow for a very restricted notion of "inheritance": For an attribute, one or (in case of `NMTOKENS` and `IDREFS`) more values[3] can be specified, either as `FIXED` or as "soft" defaults.

Literal Defaults. Literal defaults (i.e., attributes declared as `ID`, `CDATA`, or `NMTOKEN(S)`) define default values for attributes. The values are directly given in th DTD.

We use `TOKEN` defaults for describing the character of deserts: "default" deserts are hot and sandy:

```
<!ELEMENT desert (...)>
<!ATTLIST desert  ...
                  temperature NMTOKEN 'hot'
                  ground (sand|boulders|rocks|snow) 'sand'>
```

These properties become inheritable properties of the class desert (note again the handling of enumeration types as classes of objects):

desert[temperature•→"hot"; ground•→sand].

Reference Defaults. `IDREF` defaults are more complicated: there, the schema contains not directly a value, but a reference to an object/element which has to be provided by the XML instance – thus, it is not known in the DTD.

In our example, we add to every border a justice property which yields the organization which should be contacted to settle conflicts. In most cases, it is the *United Nations*, but e.g., african countries would perhaps prefer the *Organization of African Unity*; so we also model it as a soft default:

```
<!ELEMENT country (..., border*, ...)>
<!ELEMENT border EMPTY>
<!ATTLIST border  country IDREF #REQUIRED
                  length CDATA #REQUIRED
                  justice IDREF 'org-UN'>
```

[3] recall that XML represents multivalued attributes as one value

becomes

country[border⇒⇒border].
border[country→country; length→numeric; justice⇒object].

The justice default is added to internal knowledge of FLORID. Later, *in case* that an element with id org-UN – call it *un* – is found in an XML instance, the fact

border[justice•→*un*].

is added. Analogously for multivalued defaults. Note that F-Logic does not distinguish between `FIXED` and soft defaults.

5.2.7 Design Decision: Attributes vs. Subelements

Note that we decided not to distinguish attributes from subelements: both define properties which are equally accessed by path expressions of the form element.property or element..property.

+ together with the notion of *annotated literals* it allows for a representation-independent handling of properties; e.g.,

  <country name="Germany"/>

  and <country>
  <name>Germany</name>
  </country>

  yield

  germany:country[name→"Germany"]

  and

  germany:country[name→*ger-name*]
  ger-name:name[pcdata→→"Germany"]

  where *ger-name* is an annotated literal. Thus, in both cases, we have

  ?- _:country[name→N].
  N/"Germany"

– In case, an element has an attribute and one or more subelements with the same name, both accessed by the same property, e.g.,

  <country name="Germany">
  <name language="english">Germany</name>
  <name language="german">Deutschland</name>
  <name language="french">Allemagne</name>
  </country>

  yields

  germany:country[name→→{*"Germany", ger-name-eng, ger-name-fra, ger-name-ger*}].
  ger-name-eng:name[language→"english"; pcdata→→"Germany"].
  ger-name-ger:name[language→"german"; pcdata→→"Deutschland"].
  ger-name-fra:name[language→"french"; pcdata→→"Allemagne"].

  Due to annotated literals, the query then does even not bother about the distinction:

  ?- _:country[name→→N].
  N/"Germany"
  N/"Deutschland"
  N/"Allemagne"

We argue that having the same name for a literal and a subelement is (i) very seldom, and (ii) not the best design. In contrast, we expect that there are many cases where the same notions are represented by attributes in one source, and by subelements (with additional information in their attributes) in other sources. Our approach allows for smoothly integrating such sources.

Example 6 (The F-Logic database) *Here, we show an excerpt from the transformation of the* MONDIAL *database into an F-Logic database (the given mnemonic oids are only for readibility; normally* FLORID *creates internal oids):*

```
mondial[ continent↠{europe, ...}; country↠germany, ...};
         org↠{eu, nato, ...}; ...].
europe:continent[name→eur-name; area→aur-area].
eur-name:name[pcdata↠"Europe"].
eur-area:area[pcdata↠9562488].
germany: country[car_code→"D";                                  % scalar attribute
         capital→berlin;                                % scalar reference attribute
         memberships↠{eu,nato,...};                % multivalued reference attribute
         name→ger-name; population→ger-pop;                     % scalar subelements
         encompassed↠ger-enc; ethnicgroup↠{eth-ger-ger,eth-ger-tur,...};
                                                % (potentially) multivalued subelements
         border↠{border-ger-fra, border-ger-bel, ...};
         province↠{prov-bw,...}].
ger-name:name[pcdata↠"Germany"].
ger-pop:population[pcdata↠83536115].
ger-enc:encompassed[continent→europe; percentage→100].
eth-ger-ger:ethnicgroup[name→"German"; percent→95.1].
eth-ger-tur:ethnicgroup[name→"Turkish"; percent→2.3].
border-ger-fra:border[country→france; length→451].
border-ger-bel:border[country→belgium; length→167].
prov-bw:province[id→"prov-Germany-2"; name→ger-bw-name; area→ger-bw-area;
                 population→ger-bw-pop; city↠{stuttgart, ...}].
stuttgart[id→"city-germany-9"; name→ger-stgt-name; population↠{ger-stgt-pop-95}].
ger-stgt-name:name[pcdata↠"Stuttgart"].
ger-stgt-pop-95:population[year→95; pcdata→588482].
eu:organization[id→"org-EU"; seat→brussels;
                name→eu-name; abbrev→eu-abbrev;
                eu[member↠{mem-eu-member, mem-eu-appl, ...}].
eu-name:name[pcdata↠"European Union"].
eu-abbrev:abbrev[pcdata↠"EU"].
mem-eu-member:members[type→"member"; country↠{germany,france,...}].
mem-eu-appl:members[type→"applicant member"; country↠{albania,...}].
```

6 Querying an XML Database in F-Logic

The syntax of queries on the F-Logic database is now very similar to XPath/XSL Patterns (with output operators). The symbols used are different (e.g., the "/" operator of XML versus navigation by "." and "." in F-Logic), but the structure is the same. There is a generic transformation from XSL patterns to F-Logic expressions. In contrast to XSL/XQL etc., the result is not an XML document but is simply a set of variable bindings which can be used to derive additional knowledge.

Example 7 (Querying the F-Logic database: Simple queries) *The XML metadata can be queried by*

```
?- X[doctype→D].
     X/".../mondial-2.0.xml"  D/"mondial"
?- X[system↠D].
     X/"..."  D/"mondial-2.0.dtd"
?- X[systemurl↠D].
     X/"..."  D/"file:/home/.../mondial-2.0.dtd"
```

Simple navigating queries have a very similar syntax as in XPath/XSL Patterns with return operators. The queries show how our annotated literals relate with the .text() *function of XSL:*

- *Car codes of countries with the names of all cities (for countries where no provinces are known):*

```
?- _:country[car\_code->N]..city[name->CN].
XPath:  //country[@car_code?]/city/name.text()?
```

- *Names of countries with the names of all cities (for countries where no provinces are known):*

```
?- _:country[name->N]..city[name->CN].
XPath:  //country[name.text()?]/city/name.text()?
```

- *the same for countries where the provinces are known together with the province's name:*

```
?- _:country[name->N]..province[name->PN]..city[name->CN].
XPath:
//country[name.text()?]/province[name.text?]city/name.text()?
```

When following references, the object-valued properties of F-Logic allow for navigation, whereas in XPath, the id *function has to be used. Since return operators are ot yet implemented and their semantics in combination with the* id*-function is not clearly defined (we need a return operator inside the argument of the* id *function), the given XPath query syntax is rather ad-hoc:*

- *for all pairs of countries which are neighbors: the name of both countries, give the length of their common border, the population of the first one, and the name of the capital of the second one:*

```
?- _:country[name->N; population->P]
             ..border[length->L].country[name->M]
                                  .capital[name->Cap].
XPath:
  id(id(//country[name.text()? AND population.text()?]
             /border[@length?]/@country)
       [name.text()?]/@capital)
      /name.text()?
```

- *for all organizations, their name and abbreviation, and the name of their seat city.*

```
?- _:organization[name->N; abbrev->A].seat[name->SN].
XPath:
  id(//organization[name.text()? AND @abbrev?]/@seat)
    /name.text?
```

- *for all organizations, the name, and all members together with their membertype.*

```
?- _:organization[name->N]..member[type->T]..country[name->CN].
XPath:
  id(//organization[name.text()]/member[@type?]/@country)
    /name.text()
```

Example 8 (Querying the F-Logic database: Complex queries) *F-Logic also allows for more complex queries, containing e.g., aggregations, multiple joins, comparisons of derived values etc. Such queries are in general not expressible in XPath/XSL since values must be compared and communicated between different positions in the pattern.*

- *for all organizations, the sum of inhabitants of all member countries, grouped by membership types:*

```
?- _O:organization[name->N]..member[type->T],
   SP = sum{P [_O,T];
            _O..member[type->T]..country[population->P]}.
```

- *all pairs (*country, organization*) s.t.* country *is a member of the* organization *and contributes at least 25% of the population of all members of the* organization*:*

```
?- _O:organization[name->N]..member[type->"member"; country->>_C],
   SP = sum{P [_O];
            _O..member[type->"member"]
                      ..country[population->P]},
```

```
    _C[name->CN; population->CP], SP < 4 * CP.
```

- *for all organizations, their name and abbreviation, and the name of their seat city and the country where it belongs to.*

  ```
  ?- _:organization[name->N; abbrev->A; seat->C:city[name->SN]],
     _:country[city->>C].
  ?- _:organization[name->N; abbrev->A; seat->C:city[name->SN]],
     _:country..province[city->>C].
  ```

  Note that there is no property city[country$\rightarrow$...] *which could be used for determining the country where a city belongs to. In an F-Logic database, such a property can easily be added:*

  ```
  city[country=>country].
  City[country->Country] :- (Country:country)[city->>City].
  City[country->Country] :- (Country:country)..province[city->>City].
  ```

  Now, we can use the path expression

  ```
  ?- _:organization[name->N; abbrev->A]
                   .seat:city[name->SN].country[Name->CN].
  ```

6.1 Expressiveness

Data Model. The F-Logic data model covers both the DTD (by signature atoms) and the XML instance; it even holds more explicit semantics since the target classes of reference attributes are known.

Querying Language. The F-Logic querying language allows for combined querying of data and metadata. Additionally, F-Logic as a querying language is more expressive than XPath/XSL Patterns as shown above.

7 Data Integration from Multiple Sources

The transformations given in Section 3.4 provide an additional *context* argument which can be used for distinguishing different sources:

- u.parse@(dtd, *context*),
- u.parse@(sgml, *references*, *context*),
- u.parse@(xml, *references*, *ordering*, *schema*, *context*).

If *context* is non-nil, it is interpreted as a *namespace* prefix which qualifies all classes and properties defined when transforming the document. When parsing a document, by u.parse@(sgml, ..., *context*) or u.parse@(xml, ..., *context*), the parse-tree or database view, respectively, is augmented by qualified classes and properties:

- classes: every tag defines a class
  o:(*context*.class),
- properties: subelements and attributes define properties
  (*context*.class$_1$)[(*context*.method)$\Rightarrow\!\!\!\Rightarrow$(*context*.class$_2$)].
  o[(*context*.method)$\rightarrow$$value$].
  Note that basic classes, i.e., string, numeric, literal, or object, are not augmented:
  (*context*.class)[(*context*.method) $\Rightarrow\!\!\!\Rightarrow$basic_class)]
- document structure: the hierarchy is represented by
  $node$[*context*.child@(method,index)$\rightarrow$$value$].

Using namespaces, several sources can be mapped into the database, generating objects of qualified classes. Then, "matching" objects can be *fused* into an object of a common class, i.e., making one object of them which acquires all properties from both original objects. Due to the use of namespaces, the origin of the properties can still be traced back and additional steps can be done for deriving the properties the fused object should have in the final database.

Example 9 (Integration) *Consider that we access countries from two sources, the CIA World Factbook [CIA], and the* GlobalStatistics *[Glo] source (using their XML representations [Mon]):*

cia.parse@(xml, *references*, nil, dtd, cia).
gs.parse@(xml, *references*, nil, dtd, gs).

Parsing with DTD access, the following signature is generated

```
cia.continent :: node [(cia.name) => (cia.name);
                       (cia.id) => (literal)].

cia.country :: node.
cia.country :: node [(cia.name) => (literal);
  (cia.id) => (literal);
  (cia.capital) => (literal);
  (cia.government) => (literal);
  (cia.indep_date) => (literal);
  (cia.gdp_total) => (literal);
  (cia.inflation) => (literal);
  (cia.gdp_serv) => (literal);
  (cia.gdp_ind) => (literal);
  (cia.gdp_agri) => (literal);
  (cia.infant_mortality) => (literal);
  (cia.population_growth) => (literal);
  (cia.population) => (literal);
  (cia.total_area) => (literal);
  (cia.datacode) => (literal);
  (cia.ethnicgroups) =>> (cia.ethnicgroups);
  (cia.religions) =>> (cia.religions);
  (cia.languages) =>> (cia.languages);
  (cia.borders) =>> (cia.borders);
  (cia.coasts) =>> (cia.coasts)].

cia.name :: node [pcdata => (literal)].

cia.ethnicgroups :: node [pcdata => (literal);
                          (cia.name) => (literal)].

cia.religions :: node [pcdata => (literal);
                       (cia.name) => (literal)].

cia.languages :: node [pcdata => (literal);
                       (cia.name) => (literal)].

cia.borders :: node [pcdata => (literal);
                     (cia.country) => (object)].

cia.coasts :: node [pcdata => (literal)].

gs.continent :: node [(gs.name) => (gs.name);
                      (gs.id) => (literal)].

gs.country :: node [(gs.continent) => (object);
  (gs.id) => (literal);
  (gs.population) => (literal);
  (gs.capital) => (object);
  (gs.name) =>> (gs.name);
```

```
  (gs.adm_divs) =>> (object);
  (gs.main_cities) =>> (object)].

gs.city :: node [(gs.continent) => (object);
  (gs.id) => (literal);
  (gs.population) => (literal);
  (gs.name) =>> (gs.name);
  (gs.adm_divs) =>> (object);
  (gs.main_cities) =>> (object);
  (gs.population) =>> (gs.population)].

gs.province :: node [(gs.name) => (literal);
  (gs.id) => (literal);
  (gs.population) => (literal);
  (gs.capital) => (object);
  (gs.area) => (literal)].

gs.name :: node [pcdata => (literal)].

gs.population :: node [pcdata => (literal);
                       (gs.year) => (literal)].
```

Then, rules can be given which fuse identical objects of both sources:

C1 = C2:continent :-C1:(cia.continent)[(cia.name)→N2],
C2:(gs.continent)[(gs.name)→N2], substr(CN1,CN2).

C1 = C2:country :- C1:(cia.country)[(cia.name)→N],
C2:(gs.country)[(gs.name)↠N].

C1 = C2:country :- C1:country[(gs.continent)↠CT1].capital[name@(gs)↠N],
C2:country[(cia.continent)↠CT1]; capital@(cia)→N].

8 XML Schema

In Section 5, the database schema has been generated based on a DTD. XML Schema [XML99] provides a formalism for more sophisticated schema descriptions. Especially, elements of the same element type (in the XML document) can be of different XML Schema element types, having different attribute and subelement specifications.

Example 10 (XML Schema) *In our example, the country and city elements have different types of population subelements: for countries, the population is PCDATA without attributes, and for cities, it is PCDATA with a year attribute:*

```
<country car_code="D" >
  ...
  <population>83536115</population>
  ...
</country>

<city name="Berlin">
  ...
  <population year="95">3472009</population>
  ...
</city>
```

In the DTD, this cannot be described; we chose the following specification which covers both cases:

```
<!ELEMENT population (#PCDATA)>
<!ATTLIST population year CDATA #IMPLIED>
```

In contrast, an XML Schema specification allows to have different types for population subelements:

```
<complexType name="mondialtype">
   <element name="country"
            type="countrytype"
            minOccurs="0" maxOccurs="*" >
   <element name="city"
            type="citytype"
            minOccurs="0" maxOccurs="*" >
</complexType>

<complexType name="countrytype">
   <element ref="name" />
   ...
   <element name="population"
            type="integer"
            minOccurs="0" maxOccurs="1" />
</complexType>

<complexType name="citytype">
    <element ref="name" />
    <element name="population" minOccurs="0" maxOccurs="*" >
      <complexType base="integer" derivedBy="extension" >
         <attribute name="year" type ="date" use="optional"/>
      </complexType>
    </element>
    ...
</complexType>
```

Thus, the schema representation in F-Logic is

country[population⇒countrytype.complextype@(population)].
countrytype.complextype@(population)[pcdata⇒integer].
city[population⇒citytype.complextype@(population)].
citytype.complextype@(population)[year⇒integer].
citytype.complextype@(population)[pcdata⇒integer].

If such an extended schema description is available (e.g., generated by the XML Schema wrapper [DB00]) when

url.parse@(xml, _ , _, schema, _).

is called, it is used for mapping an XML instance to objects, properties and class memberships.

9 Database Export in F-Logic and XML

For using FLORID in an environment with other database systems, data export functionality is required. FLORID provides data export in F-Logic and XML format, on two different semantical levels:

- unfiltered "dump": The whole contents of the object manager is dumped, either in F-Logic or in XML format. This functionality is provided by the object manager and implemented on the frame level:
  - sys.theOM.dump@("*mode*") where $mode \in$ {flogic, xml} outputs to the current output channel,
  - sys.theOM.dump@("flogic", "*filename*") and sys.theOM.dump@("xml", "*filename*", "*doctype*") output to the given file.

- filtered export: for a given signature, only the part covered by this signature is exported. The functionality is provided by the Object Manager Access and implemented on a semantical level, i.e., based on F-Logic atoms:
  - sys.theOMAccess.export@("*mode*") where $mode \in$ {flogic, sig, xml} outputs to the current output channel,
  - sys.theOMAccess.export@("*mode*", "*filename*") where $mode \in$ {flogic, sig} and sys.theOMAccess.export@("xml", "*filename*", "*doctype*") output to the given file.

XML output to the output channel is done without a document type; thus, it can be used for combining output from several runs in a common file via output redirection.

When XML output is written to a file, the XML prolog and XML root element is generated. Here, the element name of the root element is given by the *"doctype"* argument. If the database contains an atom

"doctype"[system→*string*] or *"doctype"*[public→*string*] ,

the `<!DOCTYPE ...>` information is also written.

Example 11 (XML Output) *The XML output functionality includes the generation of a valid XML prolog:*

?- sys.theOMAccess.export@("xml", "mondial.xml", "Mondial").

generates a file mondial.xml of the form

```
<?xml version="1.0" encoding="UTF-8"?>
<Mondial>
    ...
</Mondial>
```

When additionally the DTD of the generated document is known, the prolog also contains a document type declaration:

"Mondial":doctype[system→"mondial.dtd"].
?- sys.theOMAccess.export@("xml", "mondial.xml", "Mondial").

generates a file mondial.xml of the form

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE "Mondial" SYSTEM "mondial.dtd">
<Mondial>
    ...
</Mondial>
```

Note: Even if the signature contains also atoms *doctype*[...⇒...], objects x : *doctype* are *not* output.

Example 12 *For a DTD/XML source of the form*

```
<!ELEMENT mondial (country*)>
<!ELEMENT country (...)>

<!DOCTYPE mondial ...>
<mondial>
  <country name="Germany"> ... </country>
  ...
<mondial>
```

the following F-Logic representation is generated:

mondial[country⇒⇒country].
country[...].
mon:mondial[country→→{germany, ...}].
germany:country[...].

Exporting with export@("xml", "tmp", "mondial") *generates a file* tmp *which automatically contains a surrounding* <mondial> *tag, which must* not *contain another* <mondial> *tag of the* mon *element, but only the other elements.*

9.1 Output Strategies

We will first clarify the set of atoms to be considered by each of the output modes.

Dump. The complete object manager is dumped objectwise. For every object o,

- all subclass relationships o :: c,
- all class memberships o : c,
- all slots, i.e., all atoms having o as host object

are output. Additionally, all predicate atoms are output.

Qualified Export. For a given set of signature atoms, all information covered by this signature is exported:

Definition 1 *Let S be a set of F-Logic signature atoms. Let c be an object. Then, c is a* relevant class wrt. *S if*

- *c occurs in any signature atom at host position.*

Definition 2 *Let S be a set of F-Logic signature atoms. For every object o, the following atoms are* relevant on the data level wrt. *S:*

- *all subclass relationships o :: c s.t. c is a relevant class wrt. S,*
- *all class memberships o : c s.t. c is a relevant class wrt. S,*
- *all slots $o[m@(o_1,\ldots,o_n) \rightsquigarrow v]$, s.t. there are classes c_i and a signature atom $c[m@(c_1,\ldots,c_n) \Rightarrow\!\!> c_{n+1}]$ s.t. $o : c$, $o_i : c_i$ for all argument positions i, and $v : c_{n+1}$.*

Predicates are not relevant.

Signature Export. All data describing the schema level of an F-Logic database is exported.

Definition 3 *Let S be a set of F-Logic signature atoms. For every object c which is a relevant class wrt. S, the following atoms are* relevant on the schema level wrt. *S:*

- *all subclass and membership relationships $c_1 :: c_2$ and $c_1 : c_2$ s.t. c_1 and c_2 are relevant classes,*
- *all slots $c[m@(c_1,\ldots,c_n) \Rightarrow\!\!> c_{n+1}]$.*

Predicates are not relevant.

9.2 Output Formats

FLORID provides output in F-Logic and XML format. In both cases, the output is done *objectwise* (from the internal FLORID point of view, framewise). Let o be an object; for dump, all atoms are considered, for export, only the relevant atoms are considered.

F-Logic. In F-Logic format, the objects are output as follows.

- o :: c. for all atoms o :: c, (subcl)
- o : c. for all atoms o : c, (isa)

- $o[m@(o_{1,1},\ldots,o_{1,n_1})\rightarrow v_{1,i_1+1};$

$$\vdots \qquad \text{(funcData)}$$

$$m@(o_{k,1},\ldots,o_{k,n_1})\rightarrow v_{k,i_k+1};$$
$$m@(o_{k+1,1},\ldots,o_{k+1,n_{k+1}})\twoheadrightarrow\{v_{k+1,n_{k+1}+1},\ldots,v_{k+1,i_{k+1}+m_{k+1}}\};$$

$$\vdots \qquad \text{(mvdData)}$$

$$m@(o_{k+l,1},\ldots,o_{k+l,n_{k+l}})\twoheadrightarrow\{v_{k+l,n_{k+l}+1},\ldots,v_{k+l,i_{k+l}+j_{k+l}}\};$$
$$m@(o_{k+l+1,1},\ldots,o_{k+l+1,n_{k+l+1}})\Rightarrow (v_{k+l+1,n_{k+l+1}+1},\ldots,v_{k+l+1,i_{k+l+1}+j_{k+l+1}});$$

$$\vdots \qquad \text{(funcSig)}$$

$$m@(o_{k+l+m,1},\ldots,o_{k+l+m,n_{k+l+m}})\Rightarrow (v_{k+l+m,n_{k+l+m}+1},\ldots,v_{k+l+m,i_{k+l+m}+j_{k+l+m}});$$
$$m@(o_{k+l+m+1,1},\ldots,o_{k+l+m+1,n_{k+l+m+1}})\Rightarrow\!\!\Rightarrow (v_{k+l+m+1,n_{k+l+1}+1},\ldots,v_{k+l+m+1,i_{k+l+m+1}+j_{k+l+m+1}});$$

$$\vdots \qquad \text{(mvdSig)}$$

$$m@(o_{k+l+m+p,1},\ldots,o_{k+l+m+p,n_{k+l+m+p}})\Rightarrow\!\!\Rightarrow (v_{k+l+m+p,n_{k+l+m+p}+1},\ldots,v_{k+l+m+p,i_{k+l+m+p}+j_{k+l+m+p}})]$$

Note that only in the dump@("flogic") output mode, all the above kinds of properties are output.
- in export@("flogic") mode, only (subcl), (isa), (funcData), and (mvdData) are output,
- in export@("sig") mode, only the atoms which are relevant as defined in Definition 3 are output.

XML When generating XML, we have first to define a mapping from F-structures to XML documents. Here, we choose a straightforward mapping to a *flat* XML model which is based on the idea that
- every F-Logic (object) frame is mapped to an XML element.

There are several direct consequences:
- the element has *one single elementtype* which has to cover the notion of a class in F-Logic.
- slots have to be represented by attributes or subelements with `PCDATA` (or `EMPTY`) contents, that means,
- relationships are represented by reference attributes.

In detail, we chose the following mappings:
- We assume that every object has a lowest class. This class determines the element type.
- superclass relationships are ignored,
- signature atoms are ignored,
- properties without arguments
  - are represented by attributes (except one case, see below).
    - Literal-valued scalar properties become `NMTOKEN` attributes,
    - numeric-valued multivalued properties become `NMTOKENS` attributes,
    - object-valued scalar properties become `IDREF` attributes,
    - object-valued multivalued properties become `IDREFS` attributes,
  - string-valued multivalued methods are represented by subelements since strings can contain whitespace, so an `NMTOKENS` representation is not possible. Here, every individual result is mapped to an individual subelement, e.g.,

    jfk:person[lastname→"Kennedy"; firstname↠{"John", "Fitzgerald"}]

    becomes

    ```
    <person lastname="Kennedy" id="jfk">
        <firstname>John</firstname>
        <firstname>Fitzgerald</firstname>
    </person>
    ```

- properties with arguments are mapped to subelements. Similar to multi-valued string-valued properties, every individual result is mapped to an own subelement[4].
  - arguments become attributes. If signature atoms are available, the argument classes determine the name of the attributes, otherwise they are enumerated as arg_1,...,arg_n.
  - literal-valued properties are mapped to subelements with PCDATA contents, e.g.,

    city[name⇒string; population@(year)⇒numeric].
    berlin:city[name→"Berlin"; population@(1995)→3472009].
    becomes
    <city name="Berlin" id="berlin">
    <population year="1995">3472009</population>
    </city>

  - object-valued properties are mapped to EMPTY subelements with an additional reference attribute representing the result:

    country[name⇒string; capital@(year)⇒city].
    germany:country[name→"Germany"; capital@(1990)→*bonn*;
    capital@(1995)→*berlin*].
    becomes
    <country name="Germany" id="germany">
    <capital year="1990" city="bonn"/>
    <capital year="1995" city="berlin"/>
    </country>

Note that, when parsing the XML output into FLORID again, parameterized methods are mapped to mon-parameterized methods which result in ordinary complex objects (if the original method was object-valued) or annotated Literals (if the original method was literal-valued).

References

[CB00] R. Cattell and D. Barry. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[CIA] CIA World Factbook. `http://www.odci.gov/cia/publications/factbook/index.html`.

[DB00] R. Danaei-Boroumand. Abbildung von XMLSchema nach FLORID-Schemata. Studienarbeit, Universität Freiburg (in german), 2000.

[Glo] `http://www.stats.demon.nl/`.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–612, 1998.

[May99a] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[May99b] W. May. Modeling and Querying Structure and Contents of the Web. In *Workshop on Internet Data Modeling (IDM'99), Proc. DEXA 99 Workshops*, pp. 721–725. IEEE Computer Science Press, 1999.

[MHLL99] W. May, R. Himmeröder, G. Lausen, and B. Ludäscher. A Unified Framework for Wrapping, Mediating and Restructuring Information from the Web. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM)*, number 1727 in LNCS, pp. 307–320, 1999.

[4] we could also map all results for the same arguments to a common subelement, similar to above. Both yields the same results for all set-oriented XPath queries.

[ML00] W. May and G. Lausen. Information Extraction from the Web. Technical Report 136, Universität Freiburg, Institut für Informatik, 2000. Available from `http://www.informatik.uni-freiburg.de/~dbis/Integration/`.

[Mon] The MONDIAL Database. `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[XML99] XML Schema Part 2: Datatypes. `http://www.w3.org/TR/xmlschema-2`, 1999.