Integration of XML Data in XPathLog

Wolfgang May
Institut für Informatik, Universität Freiburg, Germany
may@informatik.uni-freiburg.de

Abstract

XPathLog is a logic-based language for manipulating and integrating XML data. It extends the XPath query language with Prolog-style variables. Due to the close relationship with XPath, the semantics of rules is easy to grasp. XPathLog defines a semantics for XPath expressions in rule heads, *declaratively* specifying how to create and update XML trees and nodes.

In this paper, we show how XPathLog can be used to manipulate and restructure a database containing several XML trees. By linking subtrees, fusing elements and defining synonyms, data can be restructured and integrated into result trees. We illustrate the practicability of the approach by excerpts of a case study done with the LoPiX system.

1 Introduction

XML has been designed and accepted as *the* framework for semi-structured data where it plays the same role as the relational model for classical databases. Specialized languages are available for XML querying, e.g., XML-QL [DFF+99], Quilt/XQuery [CRF00, XQu01] and for transformations of XML sources, e.g., XSLT [XSL99] (also XML-QL and XQuery can be used for generating new XML documents from given ones since their output format is XML), but yet none of them can be seen as an XML data *manipulation* language. For writing applications for creating and manipulating XML data, the dominating language is Java, regarding the DOM model as a data structure where applications are built on. A proposal for updating constructs for XML languages has been recently presented in [TIHW01].

We propose a declarative, Prolog-style language for manipulation and integration of XML documents. The syntax and querying semantics is based on XPath [XPa99]. Whereas XSLT, XML-QL, and Quilt/XQuery use XML patterns for generating output (with the consequence that their output can only *generate* XML, but it cannot be used for *manipulating* an existing XML instance), our language deviates from these approaches: XPath-based syntax is used for querying (rule bodies) *and* generating/manipulating the data (rule heads). Also in contrast to the XML mainstream (i.e., the DOM and XML Query Data Model [XMQ01]), XPathLog works on an abstract, edge-labeled graph-based data model as an internal representation of the current XML database. This design decision has been motivated by the experiences with F-Logic/FLORID in information integration. The data model is especially tailored to data integration, allowing to re-link elements into multiple overlapping trees, fusing elements, and introducing synonyms for subelement and attribute names (note that XML-QL is also based on a graph data model, influenced by the STRUDEL project [FFLS97]). In the present paper we keep the formal, theoretical part in the background (details can be found in [May01b]), focusing on an intuitive presentation.

Structure of the paper. After reviewing XPath as the base of our language, we introduce the querying part of XPathLog in Section 2. Section 3 gives an idea of *X-structures* which serve both as the theoretical foundation and as the internal representation of the XML database. Section 4 defines the semantics of XPathLog rules, focusing on the semantics of rule heads for generating and modifying XML data. Section 5 describes the operations and strategies for XML Data integration by exploiting the flexibility of the internal data model. Section 6 gives a perspective how XML data, XML metadata (DTDs and XML Schema) and ontologies can be handled and combined uniformly in this framework. Section 7 concludes the paper.

2 XPathLog

2.1 XPath

XPath [XPa99] is the common language for addressing node sets in XML documents. It is based on navigation through the XML tree by *location paths* of the form //*step*/*step*/.../*step*. Every *location step* is of the form *axis*::*nodetest*[*filter*]*, denoting that navigation goes along the given axis. Along the chosen axis, all elements which satisfy the *nodetest* (the *nodetest* specifies the nodetype or an elementtype which nodes should be considered) are selected. From these, the ones qualify which satisfy the given *filter(s)* (applied iteratively). Starting with this (local) result set, the next location step is applied (for details, see [XPa99], or the formal semantics given in [Wad99]). The most frequently used axes are abbreviated as *path*/nodetest for *path*/child::nodetest, *path*/@nodetest for *path*/attribute::nodetest, and *path*//nodetest for *path*/descendant-or-self/child::nodetest.

Example 1 (XML, XPath) *Consider the following excerpt of a geopolitical database (an XML representation of the CIA World Factbook [CIA]) for illustrations.*

DTD:

```
<!ELEMENT cia (continent+, country+)>
<!ELEMENT continent EMPTY>
    <!ATTLIST continent id ID #REQUIRED    name CDATA #REQUIRED>
<!ELEMENT country (ethnicgroups*, religions*, languages*, borders*)>
    <!ATTLIST country    name CDATA #REQUIRED      car_code ID #REQUIRED
                         continent IDREF #REQUIRED area CDATA #IMPLIED
                         population CDATA #IMPLIED capital CDATA #REQUIRED>
<!ELEMENT ethnicgroups (#PCDATA)>   <!ATTLIST ethnicgroups name CDATA #REQUIRED>
(similar for religions and languages)
<!ELEMENT borders (#PCDATA)>      <!ATTLIST borders country IDREF #REQUIRED>
```

Excerpt of the instance:

```
<cia>
<continent id='europe' name='Europe'/>
<country car_code='CH' capital='Bern' continent='europe' name='Switzerland'
         area='41290' population='7207060' >
  <religions name='Roman Cath.'>47.6</religions>  <religions name='Protestant'>44.3</religions>
  <languages name='French'>18</languages>         <languages name='German'>65</languages>
  <languages name='Italian'>12</languages>        <languages name='Romansch'>1</languages>
  <borders country='A'>164</borders>              <borders country='F'>573</borders>
  <borders country='D'>334</borders>              <borders country='I'>740</borders>
```

```
  <borders country='FL'>41</borders>
</country>
</cia>
```

Consider the country *element only. The XPath expression*

//country[@name]/languages[@name=“German”]

returns all elements which contain information on german language in a country. Note that it is not possible to output also the name of the country using XPath.

XPath is only an addressing mechanism, not a full querying language like, e.g., the SQL querying construct. It provides the base for most XML querying languages, which extend it with their special constructs (e.g., functional style in XSLT, and SQL/OQL style (e.g., joins) in Quilt/XQuery). In the case of XPathLog, the extension feature are Prolog style variable bindings and rules.

2.2 XPathLog: Adding Variable Bindings to XPath

In Logic Programming languages, instead of a result set, for every match, a tuple of *variable bindings* is returned which can be used in the rule head. We extend the XPath syntax with the Datalog-style variable concept (and with implicit dereferencing). The variables are bound to the names/nodes/literals (for i.e., CDATA or NMTOKENS attributes) which result from the respective match; the formal semantics is based on that of XPath given in [Wad99].

Definition 1 (Reference Expressions)

- *An* XPath-Logic reference expression *is an XPath* AbsoluteLocationPath *(we refer to the numbering in [XPa99, Ch. 2]) where the XPath syntax is extended as follows:*
  - *In XPath-Logic steps,* $axis :: nodetest$ *may be replaced by* $axis :: nodetest \to V$*,* $axis :: V$*, or* $axis :: V \to W$ *where* V *and* W *are variables:*

```
[4] Step ::= AxisSpecifier "::" NodeTest Predicate*
           | AxisSpecifier "::" NodeTest "->" Var Predicate*
           | AxisSpecifier "::" Var Predicate*
           | AxisSpecifier "::" Var "->" Var Predicate*
```

- *XPath-Logic* LocationPaths *may begin with constants or variables:*

```
[2b] ConstantLocationPath ::= constant "/" RelativeLocationPath
                            | variable "/" RelativeLocationPath
```

Definition 2 (XPathLog) *Atoms are the basic components of XPathLog rules:*

- *an* XPathLog atom *is either a* LocationPath *or a* ConstantLocationPath*, or a predicate expression over these.*
- *an XPathLog atom is* definite *if it uses only the child and sibling axes and does not contain negation, disjunction, function applications, and* proximity position predicates *(i.e., does not use the* position() *and* last() *functions). These atoms are allowed in rule heads (see Section 4.2); the excluded features would cause ambiguities what update is intended.*
- *an* XPathLog literal *is an atom or a negated atom,*
- *an* XPathLog query *is a list* $?\!-L_1, \ldots, L_n$ *of literals (in general, containing free variables),*
- *an* XPathLog rule *is a formula of the form*

  $A_1, \ldots, A_k \leftarrow L_1, \ldots, L_n$

where L_i are literals and A_i are definite atoms. Note that in contrast to usual Logic Programming, we allow for lists of atoms in the rule head which are interpreted as conjunctions.

XPath-Logic combines first-order logic and reference expressions. In this paper, we consider only the Horn fragment of XPath-Logic, i.e., logical rules over reference expressions and predicates over them (for full XPath-Logic, see [May01b]).

Example 2 (XPathLog) Pure XPath expressions: *pure XPath expressions (i.e., without variables) are interpreted as* existential queries *which return* true *if the result set is non-empty:*

?- //country[@name = "Switzerland"]//languages[@name="German"]/text().
true

since the country element which has a name attribute "Switzerland" contains at least one language descendant with a name attribute "German" and non-empty text contents.

Output Result Set: *The query "?- $xpath$→N" for any xpath binds N to all nodes belonging to the result set of $xpath$:*

?- //country[@name = "Switzerland"]//languages[@name="German"]/text()→P.
P/65

respectively, for a result set consisting of elements, logical ids are returned:

?- //country[@name = "Switzerland"]//languages[@name="German"]→L.
L/*lang-ch-german*
:

Additional Variables: *XPathLog allows to bind all nodes which are considered by an expression (both by the access path to the result set, and in the filters):*
The following expression returns all tuples (C, L, P) such that language L is spoken by P percent of the people in a country C with more than 1000000 inhabitants:

//country[@name→C and @population>1000000]/languages[@name→L]/text()→P.

C/'Austria'	L/'German'	P/100
C/'Belgium'	L/'French'	P/32
C/'Belgium'	L/'German'	P/1
:		

Dereferencing: *Give every pair of names (N_1, N_2) of countries C_1, C_2 such that C_1 and C_2 are neighbors, and C_1 is larger than C_2:*

?- //country[@name→N1 and @area→A_1]/borders/@country[@name→N2 and @area→A_2],
A_2 > A_1.

Navigation Variables: *Are there elements which have a name attribute with value "German", and of which type are they?*

?- //Type→X[@name="German"].

Type/languages	X/*lang-D-german*
Type/ethnicgroups	X/*ethngrp-D-german*
Type/languages	X/*lang-CH-german*
:	

Schema Querying: *The use of variables at name positions further allows schema querying, e.g., to give all names of subelements of elements of type* country *which have a* name *attribute:*

```
?- //country/N/@name.
N/languages
N/ethnicgroups
N/religions
:
```

The schema querying functionality can also be used for validation wrt. a DTD or a given XML Schema (which can be present as an XML tree in the same XPathLog database, see Example 10).

Further examples can be found and executed with the LoPiX system [LoP].

3 Data Model

As already stated, XPathLog works on an abstract, edge-labeled graph-based data model (similar to the XML-QL data model) as an internal representation which supports an efficient representation of a forest of *overlapping* XML trees. The data model is especially tailored to the requirements of data integration, (e.g., re-linking elements into multiple overlapping trees, element fusion, synonyms as described in Section 5). The full theory can be found in [May01b]; here we directly define the underlying X-structures, following the DOM idea for representing XML instances. The main features of the model are:

- the universe consists of the element nodes of the XML instance and literals used in attribute values and text contents;
- elements have properties: (i) subelements (ordered) and (ii) attributes (unordered);
- multivalued attributes (`NMTOKENS` and `IDREFS`) are silently split;
- reference attributes are silently resolved.

Definition 3 (X- Structure) *For a given XML instance, the universe consists of a set $\mathcal{N}$ of names (i.e., element and attribute names), a set $\mathcal{V}$ of nodes, and a set $\mathcal{L}$ of literals. An X-structure $\mathcal{X}$ then consists of*

- *an interpretation of predicates over $\mathcal{N}$, $\mathcal{V}$, and $\mathcal{L}$, and*
- *a mapping which associates with every $x \in \mathcal{V}$ two lists of pairs, representing the child and attribute axes:*
  - $\mathcal{A}_{\mathcal{X}}(\textsf{child}, x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})^{\mathbb{N}}$ *and*
  - $\mathcal{A}_{\mathcal{X}}(\textsf{attribute}, x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})^{\mathbb{N}}$ *(arbitrary enumeration; recall that reference attributes are resolved silently into references to nodes).*

There is a canonical mapping from XML instances to X-structures. The X-structure contains only the basic facts about the XML tree. For the other axes, $\mathcal{A}_{\mathcal{X}}(axis, x)$ is derived from $\mathcal{A}_{\mathcal{X}}(\textsf{child}, x)$ according to the XML specification. Note that the attribute and element names in $\mathcal{N}$ are full citizens of the language (which e.g., may occur in predicates and can be bound to variables).

Although it "looks like" DOM, the data model significantly differs from the DOM and XML Query Data Model [XMQ01]: The data model does not impose any additional requirements on $\mathcal{A}_{\mathcal{X}}(\textsf{child}, \_)$ – thus,

- elements may have more than one parent, i.e., belong to several trees (or, *tree views*),
- the structure may be cyclic,

- there is no global order of elements, but the children relationships are ordered for each individual element.

Thus, the internal model is not a forest, but a graph which *covers* the "pure" XML tree data model. In this graph, each node n of the database is a potential root element for a *tree view* which recursively consists of subelements and attributes. The subtree rooted an element r is a "useful" XML instance if

- the subelement relation in the subtree is acyclic, and
- the targets of outgoing references from the subtree also belong to the subtree.

Nevertheless, also nodes which do not satisfy these conditions may be used as roots if the subtree is pruned by a suitable projection to a result signature (see Section 5).

4 XPathLog Rules

XPathLog rules are logical rules over XPathLog expressions (where we additionally allow conjunctions in the rule head).

4.1 Right Hand Side: Evaluation of Queries

The semantics of XPathLog queries wrt. an X-structure is defined by operators $\mathcal{S}$ and $\mathcal{Q}$ derived from the formal semantics given in [Wad99]. $\mathcal{S}$ evaluates reference expressions and returns an *annotated result list*, i.e.,

- a result list, and
- with every element of the result list, a list of variable bindings (answers) is associated.

$\mathcal{Q}$ evaluates formulas and filter predicates and returns a set of variable bindings, similar to Datalog. The formal definition of the semantics by structural induction can be found in [May01b]. Here, we restrict ourselves to an example:

Example 3 (Semantics) *Let $\mathcal{X}$ be the canonical X-structure to the XML instance given in Example 1, let*

$$expr := \text{//country[@car\_code}\to\text{CC and languages[@name}\to\text{N and text()}\to\text{P]]}\to\text{C}$$

$$\begin{array}{lll}
\mathcal{S}_{\mathcal{X}}(expr) = & \text{list}((ch, \{\text{CC/“CH”, N/“French”, P/“18”,} & \\
& \quad\quad \text{CC/“CH”, N/“Italian”, P/“12”,} & \\
& \quad\quad\quad\quad\quad \vdots & \}), \\
& (b, \{\text{CC/“B”, N/“French”, P/“32”,} & \\
& \quad\quad \text{CC/“B”, N/“Dutch”, P/“55”,} & \\
& \quad\quad\quad\quad\quad \vdots & \}), \\
& \quad \vdots &)
\end{array}$$

Thus, the evaluation of the rule body results in a set of variable bindings which are propagated to the head where facts are added to the model.

4.2 Left Hand Side

Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or Quilt/XQuery where the structure to be *generated* is always specified by XML patterns (this implies that these languages do not allow for updating existing nodes – e.g., adding children or attributes –, but only for generating complete nodes).

In contrast, in XPathLog, existing nodes are communicated via variables to the head, where they are *modified* when appearing at host position of atoms.

The head of an XPathLog rule is a set of definite XPathLog atoms (cf. Definition 2). When used in the head, the "/" operator and the "[...]" construct specify which properties should be added or updated (thus, "[...]" does not act as a filter, but as a *constructor*). Recall that for the left hand side, proximity position predicates are not allowed; instead the position where a child or sibling should be inserted can be specified by

$host[axis(i) :: name \rightarrow value]$

where axis is either child or a sibling axis. If no position is given, the new element is appended at the end of the axis.

The global semantics of XPathLog programs is defined by bottom-up evaluation based on a T_P operator similar to Datalog. Atoms in the rule head are resolved ("*atomized*") into atoms of the form n[$axis$::e→w] and predicates over constants and variables which immediately define the extensions to the underlying X-structure (again, the formal definition of atomize and of the T_P operator which evaluates rules and extends $\mathcal{X}$ with the resulting facts can be found in [May01b]).

Example 4 (Atomization) *The query*

?- //country[@name→N1 and area→_A1]/borders/@country[@name→N2 and area→_A2],
A_2 > A_1.

is atomized into

?- root[descendant::country→_C1], _C1[@name→N1], _C1[@area→_A1], _C1[borders→_B],
_B[@country→_C2], _C2[@name→N2], C2[@area→_A2], _A2 > _A1.

Creation of Elements. Elements can either be created as *free* elements by atoms of the form /*name*[...] (meaning "some element of type *name*" – in the rule head, this is interpreted to create an element which is not a subelement of any other element), or as subelements.

Example 5 *We create a new (free) country element with some properties:*

/country[@car_code→"BAV" and @capital→X and city→X and city→Y] :-
//city→X[name/text()="Munich"],
//city→Y[name/text()="Nurnberg"].

Note that the two city elements are linked *as subelements. This operation has no equivalent in the "classical" XML model: these elements are now children of* two *country elements.*

Modification of Elements. Already existing elements can be assigned as subelements to existing elements by using filter syntax in the rule head (as already done above). When using the child or attribute axis for updates, the host of the expression gives the element to be updated or extended; when a sibling axis is used, effectively the parent of the host is extended with a new subelement. A ground instantiated atom $n[child :: s \rightarrow m]$ makes m a subelement of type s of n; if the atom is of the form n[child(i)::s→m] or n[following-sibling(j)::s→m], this means that the new element to be inserted should be made the ith subelement of n or jth sibling of n. A ground instantiated atom of the form $n[@a \rightarrow v]$ specifies that the attribute @a of the node n should be set or extended with v. If v is not a literal value but a node, a reference to v is stored.

Example 6 *The following rule adds a (*PCDATA*) subelement* name*:*

C/name[text()→“Bavaria”] :- //country→C[@car_code=“BAV”].

Here, the atomized version of the rule head is

C[name→_N], _N[/text()→“Bavaria”] :- //country→C[@car_code=“BAV”].

The body produces the variable binding C/bavaria. *When the head is evaluated, first, the fact* bavaria[child::name→x_1] *is inserted, adding an (empty)* name *subelement* x_1 *to* bavaria *and binding the local variable* _N *to* x_1. *Then, the second atom is evaluated, generating the text contents to* x_1.

Using Navigation Variables for Restructuring. For data restructuring and integration, the intuitiveness and declarativeness of a language gains much from variables ranging not only over data, but also over schema concepts (classically, relation and columns, as, e.g., in SchemaSQL [LSS96]). Such features have already been used for HTML-based Web data integration with F-Logic [KLW95, LHL+98].

Extending the XPath wildcard concept, XPathLog also allows to have variables at name position. Thus, it allows for schema querying, and also for generating new structures dependent on the data contents of the original one. Here, XPathLog silently casts strings into names when a variable is bound to string contents in the body, and occurs at name position in the head:

Example 7

Consider another data source which provides data about waters according to the DTD

<!ELEMENT terra (water+, ...)>
<!ELEMENT water (...)> <!ATTLIST water name CDATA #REQUIRED ...>

which contains, e.g., the following elements:

<water type=“river” name=“Mississippi”> ... </water>
<water type=“sea” name=“North Sea”> ... </water> .

This tree is converted into the target DTD

<!ELEMENT geo ((river|lake|sea)*)>
<!ELEMENT river (...)> <!ATTLIST river name CDATA #REQUIRED ...>
(analogously for lakes and seas)

by the rules

```
result/T[@name→N] :- //water[@type→T and @name→N].
    % generate the elements ... and then copy attributes and contents
    % using variables at element name and attribute name position
X[@A→V] :- //water[@type→T and @name→N and @A→V], //T→X[@name→N].
X[S→V]  :- //water[@type→T and @name→N and S→V], //T→X[@name→N].
```

yielding, e.g.,

<river name=“Mississippi” ...> ...</river> and <sea name=“North Sea” ...> ...</sea>

4.3 Semantics of XPathLog Programs

An XPathLog program is a declarative specification how to create a set of XML documents from one or more input documents. The formal semantics of XPathLog programs is defined by bottom-up evaluation based on a T_P operator similar to Datalog. Thus, the semantics coincides with the usual understanding of a stepwise process. Additionally, an intuitive meaning of negation and other nonmonotonic features can easily be expressed by user-defined stratification. Similar to XSLT, the language is also understandable without theoretical knowledge about Logic Programming, simply from the understanding of rule-based manipulation of XML structures.

5 Data Integration: Handling Multiple XML Trees

The internal database is not intended to represent "the XML document", but a *database* where XML documents may be defined as "views" – some of the views are the original XML trees. In general, an X-structure may represent several trees which may be overlapping, i.e., subtrees can belong to several tree views. In the following, we focus on the modeling aspect of multiple trees for *data restructuring* and *integration*. The presented strategies show the flexibility of the XPath-Logic data model.

5.1 Tree Operations

Projection by Signature. In XPathLog, a projection is specified by a root node and a *signature* (i.e., a description which subelements and attributes belong to the view).

Generating an Isolated Result Tree. Projection is only suitable if a property should be handled homogeneously for all elements in the tree. The "opposite" strategy is to create the result tree completely from scratch by collecting nodes (in the extreme case, only literals, i.e. attribute values and text contents) and structuring them by creating new elements and subelement relationships. Building a separate tree is the recommended strategy if the structure of the result is very different from the original tree.

Generating a Result Tree by Selecting and Linking Subtrees. As a compromise between the above "extreme" strategies, a result tree can be created by *linking* subtrees of the original document to it, extending them appropriately, and projecting the result. By "reusing" subtrees, this strategy needs much less storage (and much less copying operations) than generating a completely new tree. Additionally, the linked and updated elements can be part of multiple views.

Namespaced Input and Multiple Input Trees. *Namespaces* can be used for distinguishing elements originating from different sources. This allows to distinguish different source trees (even when elements of different trees are *fused* (see below) during integration) and also possibly generating several result trees as views on the internal database. When integrating multiple input trees, namespaces are associated with *groups* of documents. Documents which semantically belong together and use the same DTD also share the same namespace. The actual decision depends on the situation, e.g., if the task consists of combining consistent sources describing different but overlapping application domains (e.g., a flight database and hotel bookings), or combining sources containing possible inconsistencies on the same application domain (e.g., integrating catalogs from different suppliers).

5.2 Merging and Fusing Operations

The edge-labeled navigation graph model with names as full citizens of the modeling allows to augment the above operations which special integration features on multiple *overlapping* trees. The integration process starts with parsing all source trees augmented with suitable namespaces. Then, the result tree is constructed based on nodes and literals of these trees by the following operations:

1. Synonyms: identifying and renaming properties
2. Element fusion: identifying elements in different sources which represent the same object in the application domain.
3. Linking and Collecting: elements and tree fragments are linked together to define a result tree view.

Synonyms. In contrast to the pure DOM model, the names are also elements of the universe which can be bound to variables, used in predicates, and especially *equated* with other names and synonyms. The latter proves very useful in data integration: When elements from a source are integrated into the result tree, in general also some of their properties should also completely become properties of the result view.

Here, synonyms are an efficient means for taking a whole property from a source tree (and namespace) to the result tree: By equating

$\mathit{namespace{:}name}_1 = \mathit{name}_2$.

the property name_2 is defined to have the same extension as the original property $\mathit{namespace{:}name}_1$. Note that this does *not* introduce new children or attribute nodes, or even relationships, but "only" defines an alternative access path.

Using this strategy, the internal database can be seen as a "two-level" model: the source(s) provide tree structures which *may* be used. The result views are then defined using parts of this structure, and extending it. In the following, the strategy is applied to multiple input trees.

Fusing Elements and Subtrees. *Fusing* elements which represent the same real-world entity from different data sources into a unified element is an important task in information integration. The result

1. is still an element of *both* source trees, i.e., positive queries against the original tree using the original namespace still yield at least the original answers,
2. collects the attributes of both original elements,
3. collects the subelements of both original elements.

(1) is easily satisfiable in XPathLog by adding appropriate links to the navigation graph. (2) does also not cause any problems: if the original elements use different namespaces, the attributes are simply collected, otherwise for the attributes which are present for both original elements, their values are accumulated. Attributes are not ordered, and also the values of multivalued attributes are not ordered. Only (3) needs a further specification, how the order of subelements is dealt with. Regarding the problem from the database point of view, this aspect can be ignored, accepting any kind of union of two lists.

Example 8 (Integration: Object Fusion) *Consider data sources with namespaces as shown in Figure 1. Both describe countries, where* cia *contains information about name, area, population and capital, and* gs *contains information about cities. An obvious and typical integration step is to unify the* countries *in the* cia *tree with the* countries *in the* gs *tree and link them to the result tree root node:*

```
result[country→C], C1 = C2 :-
        cia/cia:country→C1[@cia:name→N], gs/gs:country→C2[@gs:name→N].
```

The above rule makes the fused country a child of the result *node: For every country c which is present in both databases (there are some* cia *"countries" which are not actual countries but territories, sometimes even unpopulated), the country elements representing c in* cia *and* gs *are identified and the* *result is then an element of three trees:* cia, gs, *and* result *(cf. Figure 2). The example is continued below.*

The resulting elements are linked to the result tree, and the properties which should be contained in the result view have to be defined. This can be done either by introducing a synonym in the result namespace for a property in one of the source namespaces, or by adding appropriate links parallel to the already existing links (if a property is copied only for some of its instances).

Example 9 (Integration: Synonyms)
Consider the situation obtained in Example 8. After defining the synonyms

cia:name = name.	gs:city = city.	gs:text() = text().
cia:area = area.	gs:name = name.	
cia:population = population.	gs:population = population.	

the result tree fragment as given in Figure 2 is obtained. Adding the capital *links with*

```
C[@capital→City] :-
    result/country→C[@cia:capital→Name and city→City[name/text()=Name]].
```

completes this integration step.

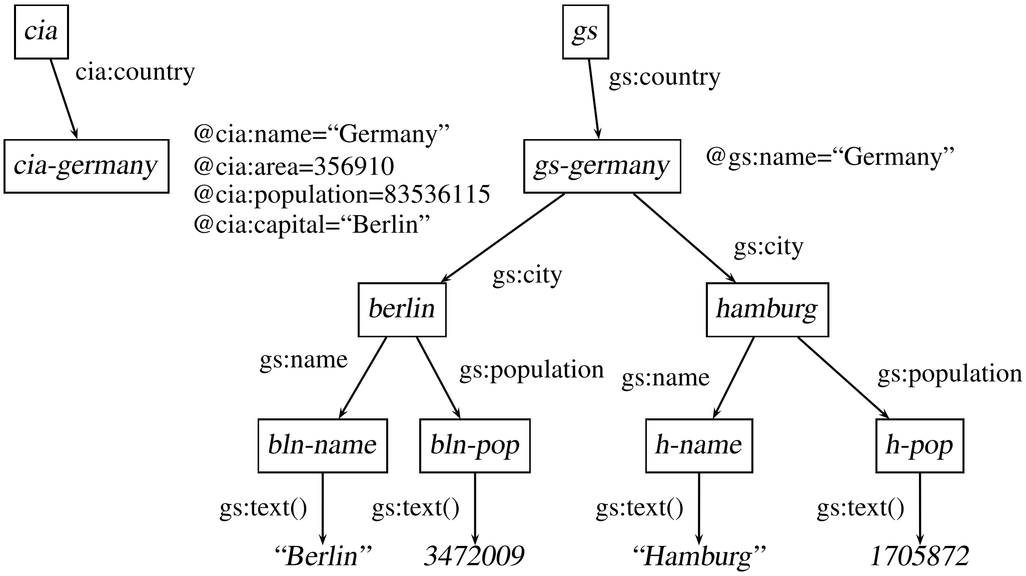

Figure 1: Element fusion – before

The above strategies of *element fusion*, *linking*, and *synonyms* allow for powerful integration concepts for generating a result tree (or even several result trees) from a set of sources. When the integration and restructuring process is completed, projections are used to define *result views* of the internal database.

A crucial feature of a data integration language is that these tasks can be specified in an intuitive, understandable way, and that the language is powerful enough to allow for short and concise statements. The DOM model is not suitable for such operations:

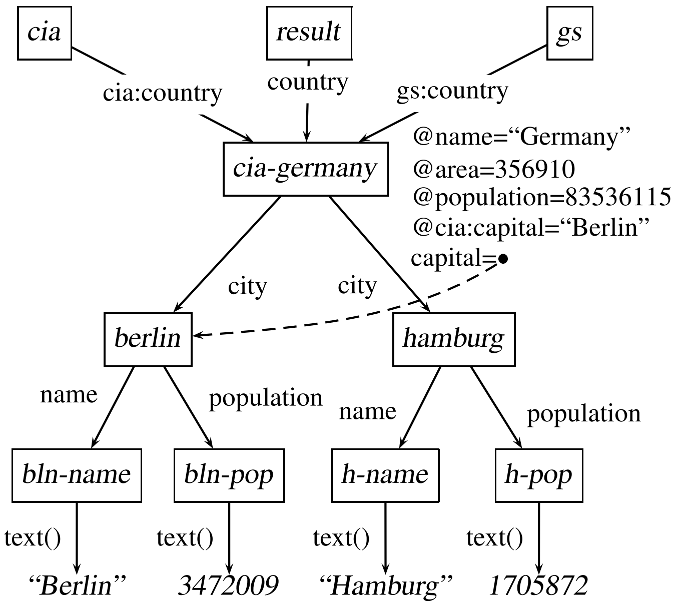

Figure 2: Element fusion – after

- every element can only have one unique parent, thus there is no way to define views as "overlays" on the source trees,
- every element has a unique name. Thus, expecially synonyms (which provide a simple, but powerful means) are not supported.

The edge-labeled data model underlying the present approach which also makes names first-class citizens of the model (and the language, respectively), supports data integration by *equating* on different levels:

- by equating elements, which represent the same real-world entity in different sources ("*fusing*" objects), these can be made a single element in the internal database, and
- by equating *names*, synonyms for properties can be defined which allows for an efficient integration of properties from several original namespaces.

The above operations and strategies are described in detail in [May01b] and have been successfully applied in the MONDIAL case study [May01a].

6 Advanced Integration Issues

In general, *metadata* is used for data integration, either merely based on schema information, e.g, in heterogeneous databases and distributed databases (projects such as TSIMMIS [GMPQ+97]), or also on *ontologies* which can be seen as databases on meta-data, giving hints how application-semantical concepts are related (e.g., InfoSleuth [BBB+97], InformationManifold [LSK95], or OBSERVER [MKSI96]). Such information can easily be added to the internal database in case that it is available in XML format.

For representing metadata, XML provides the concepts of DTDs and XML Schema [XML99]. DTDs can be transformed into an internal signature representation. Nevertheless, the DTD data does not completely become part of the XML database.

Instead, XML Schema documents are valid XML instances, thus they can be mapped directly into XML trees in the database, and associated with an entry constant. Then, the integration rules can homogeneously use the data trees and the metadata trees.

Example 10 (Combining Data and Metadata Trees)
Consider the following XML Schema specification of the cia *source:*

```
<schema xmlns ='http://www.w3.org/1999/XMLSchema' ...>
<complexType name='cia'>
    <element name='continent' type='continent_T' minOccurs='1' maxOccurs='*'/>
    <element name='country' type='country_T' minOccurs='1' maxOccurs='*'/>
</complexType>
<complexType name='continent_T' > ... </complexType>
<complexType name='country_T'>
    <attribute name='name' type='CDATA' use='required'/>
    <attribute name='continent' type='IDREF' use='required'/>
    :
    <element name='borders' minOccurs='0' maxOccurs='*' >
      <complexType base='string' derivedBy='extension'>
        <attribute name='country' type='IDREF' use='required'/>
      </complexType>
    </element>
</complexType>
```

After adding this XML tree to the database under the constant ciaSchema*, we can, e.g., validate the instance wrt. the schema, or check which are the target types of reference attributes and add them to the XML Schema tree:*

```
A[@targetElements→TTS] :-
    ciaSchema//complexType/element[@name→ENS and @type→ETS],
    ciaSchema//complexType[@name→ETS]/attribute→A[@name→ANS and @type="IDREF"],
    % now ANS is an IDREF attribute name of elements with name ENS,
    string2Object(ENS, EN), string2Object(ETS,ET), string2Object(ANS,AN),
    % two-way built-in mapping predicate, e.g., string2Object("country",country) holds
    cia//EN[@AN→Target], cia//TargetType→Target, string2Object(TTS, TargetType).
```

The above extend the internal representation of the ciaSchema *tree:*

```
<complexType name='country_T'>
    <attribute name='continent' type='IDREF' target='continent' />
    :
    <element name='borders' ... >
      <complexType ... >
        <attribute name='country' type='IDREF' target='country' />
      </complexType>
    </element>
</complexType>
```

Similarly, e.g., *links* between elements and schema elements representing element types can be added to the database.

6.1 Ontologies

Ontologies which are accessible in XML format can also be added as XML trees to the database in order to guide the integration process. Then, rules can be specified which use (i) data,

(ii) metadata like XML Schema, and (iii) additional ontology databases. Then, XPathLog rules can be used for reasoning on the meta-level, and then these results can be exploited for integrating the data given by the instances:

- Metadata + ontology: search for related concepts in different data sources, and identify concept overlappings and disjoint parts which extend each other.
- Data + results from above + graph matching algorithms: identify data overlappings (e.g., in a database on cities in european countries, and a database on economics in the G7 countries) and use them for integrating databases, also using analogy reasoning.

7 Related Work and Conclusion

Several XML querying and transformation languages have already been mentioned in the introduction. XPath provides the basis for the transformation language XSLT, which is – similar to XPathLog – rule-based, but following a functional idea. XML-QL [DFF+99] is another querying/transformation language based on matching of XML-style patterns. QUILT [CRF00] is a recent proposal for a comprehensive XML query language which also directly produces XML output. XML Data integration in an XML-QL-based environment is described [BGL+99]. The above languages are also declarative and in some sense rule-based. None of these languages allows for updating a database. They do also not provide variables at name positions, which are crucial for metadata-driven data integration. A proposal for updating XML has been presented in [TIHW01]. Since it is based on a single-parent model (according to the XML Query Data Model), it does not allow for re-linking elements (but requires a copying of subtrees which causes severe problems with maintenance of reference attributes).

Other approaches to integration of semi-structured data, especially focusing on semi-structured data as databases (in contrast to documents) are OEM/Tsimmis [GMPQ+97], STRUDEL [FFLS97], and F-Logic [KLW95, LHL+98], using semi-structured data models of the respective languages (in pre-XML times) in combination with rule-based languages.

Conclusion. To our knowledge, XPathLog is the first implemented, declarative, native XML language which allows for view definition and updates. XPathLog is completely XPath-based, ensuring that its declarative semantics is well understood from the XML perspective. Especially the nature of rule based bottom-up programming is easily understandable for XSLT practitioners, providing even more functionality. We expect that XPathLog is especially well-suited for data integration where expressive languages are needed for declaratively specifying powerful strategies, ranging over data, metadata, and meta-metadata. XPathLog has been implemented in the LoPiX system [LoP].

References

[BBB+97] R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1997.

[BGL+99] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999.

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

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *WebDB 2000*, pp. 53–62, 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference*. W3C, 1999. World Wide Web Consortium Technical Report, `www.w3.org/TR/NOTE-xml-ql`.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3):4–11, 1997.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2), 1997.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[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.

[LoP] The LoPiX System. `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[LSK95] A. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems*, 2(5), 1995.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – A Language for Interoperability in Relational Multi-Database Systems. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 239–250, 1996.

[May01a] W. May. Information Integration in XML: The MONDIAL Case Study. Technical report, 2001. Available from `http://www.informatik.uni-freiburg.de/~may/lopix/lopix-mondial.html`.

[May01b] W. May. XPath-Logic and XPathLog: A Logic-Based Approach for Declarative XML Data Manipulation. Technical report, Universität Freiburg, 2001. Available from `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[MKSI96] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *International Conference on Cooperative Information Systems (CoopIS '96)*, 1996.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Halevy, and D. Weld. Updating XML. In *ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[Wad99] P. Wadler. Two semantics for XPath. 1999. `http://www.cs.bell-labs.com/who/wadler/topics/xml.html`.

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

[XMQ01] XML Query Data Model. `http://www.w3.org/TR/query-datamodel`, 2001.

[XPa99] XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[XQu01] XQuery: A Query Language for XML. `http://www.w3.org/TR/xquery`, 2001.

[XSL99] XSL Transformations (XSLT). `http://www.w3.org/TR/xslt`, 1999.