# On an XML Data Model for Data Integration

Wolfgang May, Erik Behrends

Institut für Informatik, Universität Freiburg, Germany
{may|behrends}@informatik.uni-freiburg.de

**Abstract.** We consider the problem of integrating XML data using a warehouse strategy. In particular, we show that the DOM model and the XML Query Data Model are not suitable for data integration. We present a solution by a node-labeled graph-based data model, called *XTreeGraph*, for an internal XML *database* that represents multiple, overlapping XML trees, or *tree views*. The practicability of the approach is shown by a rule-based XML querying and manipulation language, implemented in 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] and XQuery [W3C][1] and for transformations of XML sources, e.g., XSL(T).

In this paper, we draw the consequences of thorough investigations in the areas of semistructured data and data integration. Integration of semistructured data has been dealt with already in pre-XML times, e.g., in OEM/Tsimmis [GMPQ+97], Strudel [FFLS97], and F-Logic/Florid [KLW95,LHL+98], using "proprietary" semi-structured data models of the respective languages. For taking advantage from the experiences with data integration from these projects, it is desirable to apply the developed powerful operations and strategies also to the XML area. Our work in the XML area is based on the experiences with Florid, e.g., in the Mondial case study [MHLL99] which has been carried over to XML in [May01a], using *XPathLog* [May01d] in the LoPiX system [May01b]. The investigations showed that the proposed (query) data models for XML do *not* support crucial features for updates and data integration.

Already when considering pure update functionality, it is striking that XSLT and XML-QL do not provide update constructs; only for XQuery, a proposal for updates has recently been presented in [TIHW01]; we show some problems of this approach below. For writing applications for creating and manipulating XML data, the dominating language is Java, using the DOM model (thus, also facing the restrictions as described in the following).

**Data Integration.** Data integration means completely different things in the document area and in the database area: Integration of documents mainly means generating a large tree from smaller ones without actually touching the contents

[1] all W3C XML notions, e.g., XML, DOM, XPath, XQuery, XML Query Data Model, XML Schema, and RDF can be found at this site.

of the original trees – here, transformational approaches like XSLT are suitable. In contrast, for databases, the sources provide a database where objects and their properties are described; references are regarded as object-valued properties. There is not a dominating hierarchical structure, but every source defines a densely connected graph. Thus, a tree-oriented proceeding is not suitable. A language for XML data integration must support arbitrary navigation in the XML instance, and specialized operations on objects and database fragments. Research on data integration has also pointed out the need for powerful languages that allow for combining database contents with database metadata.

XPathLog/LoPiX [May01c,May01b] is a system for XML data integration. Two variants were evaluated: the "original" one, based on an edge-labeled graph data model, called *XTreeGraph*, and a derived variant using a DOM implementation, complemented with parts of the original LoPiX storage modules that support index structures and proprietary XPathLog features [Beh01]. The comparison showed that the DOM is not suitable from the database point of view.

The paper is structured as follows: In Section 2, the common XML data model(s) are reviewed, and its constraints are analyzed. Updates on XML are considered in Section 3; Section 4 then focuses on the requirements for data integration. An alternative data model, coping with these issues is proposed in Section 5. Its realization by the XPathLog language and the LoPiX system is described in Section 6. Section 7 concludes the paper.

## 2 The XML Data Model(s)

The *abstract, informal* data model of an *XML instance* is a *tree*, consisting of *nodes*. The data model is formally described by the *DOM* and the *XML Query Data Model* – note that the "common" *representation* of XML data by ASCII files is not *the* XML Data Model. Every XML instance is associated with a unique *document node* that serves as an entry point for "accessing" the XML instance. The document node has a unique child (which is an *element node*) which is the *root node* of the document.

- *Element nodes* are the inner nodes of the tree, having a name (often referred to as their "tag", e.g., *country*, or *city*). The *element contents* is an ordered list of children, i.e., of *element nodes* and *text nodes*. Additionally, elements may have an unordered set of *attribute nodes* associated with them.
- *Text nodes* are leaves, having only text contents and no children and no attributes.
- *Attribute nodes* are also leaves, having a name (e.g., *area* or *capital*), a type, and a value. The basic XML model distinguishes between the following attribute types:
  - `CDATA`, `NMTOKEN`, `NMTOKENS`: scalar and multivalued value attributes, e.g. (`NMTOKENS`)

    <country name="Switzerland" industry="machinery chemicals watches">
  - `ID`: a distinguished scalar `NMTOKEN` attribute. Its value must uniquely identify an element throughout the whole XML instance.

- `IDREF`/`IDREFS`: scalar/multivalued *reference attribute*, its value(s) must occur as the value of an `ID` attribute, e.g.,
  <country name="Switzerland" memberships="org-efta org-oecd ...">

In general, every XML instance is of a certain *document type* i.e., built from element nodes and attributes according to a certain specification, given either by a DTD (Document Type Description) or an XML Schema description.
Based on this informal data model, two data models have formally been defined:

– the DOM (Document Object Model) as a specification of an abstract datatype for handling XML data in object-oriented languages, and
– the XML Query Data Model as a formal data model for the XML Query Algebra, e.g., underlying the formal design of XML query languages.

The formalisms DTD, XML Schema, and RDF are then used for describing the "schema" of XML tree instances (names to be used, allowed nesting etc).

### 2.1 Constraints of the Data Model

In the data model described above, XML data exists only in form of (document) trees, imposing several constraints and requirements on the applications. Most of these constraints stated in [XMQ01, Section 7] are too restrictive when dealing with data *manipulation*, and even more, when investigating data *integration*:

– Node identity: two nodes are identical if they *are* the same node. This leads to problems when trying to handle nodes originating from several document trees that are known to have the "same semantics" (below, element fusion).
– Unique parent (integration): this problem is closely related. Under this assumption, nodes originating from different documents may not be identified or fused.
– Unique parent (inserting/copying): when restructuring XML data, the elements cannot be "reused" directly (which would result in two parents) but have to be copied. Then, the maintenance of reference attributes gets difficult (see also Section 3.3).
– Unique name: every element is of a unique element type. Under this assumption, an element cannot be accessed under another name. Such functionality is e.g. useful if the target terminology differs from the source terminology (see *synonyms* in Section 4).

The above constraints are problematic for data integration, where corresponding elements in different sources have to be identified and merged, and a result tree consisting of information from the sources has to be generated. In the XML Query Data Model, this is only possible by creating the result tree from scratch without reusing elements from the sources. Since (deep-)copying of whole subtrees is expensive both wrt. computation time and storage, this is not always a favorable solution. Additionally, the maintenance of references is problematic. In Section 5, we describe a different data model that supports the reuse of elements by allowing for *overlapping* trees, elements having multiple parents, element fusion, and synonyms for names.

The handling of attributes causes another, smaller problem:

– Most of the languages do not support splitting of attribute nodes with non-atomic values (`NMTOKENS/IDREFS`), except by using explicit string operations.
– The concept of *node references* is defined, but the actual technique of handling references is not implied. In practice, references are managed by `IDREF/ID` strings. `ID`s are global to the document which also yields problems when integrating data from different documents. Explicit dereferencing is supported in XPath and XQuery, but XML-QL does not provide a construct for dereferencing `IDREFS` attributes – here, string manipulation is explicitly required for splitting into individual references which can then be used in a join.

# 3 Updating XML Data

## 3.1 XPath

XPath is the common language for addressing node sets in XML trees, based on tree navigation by *location paths* of the form document($url$)/*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 that satisfy the *nodetest* (which specifies the nodetype or an elementtype) 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.

## 3.2 Updates in XML Languages

Already for updates, the tree model causes problems. Whereas XML-QL does not provide any update constructs, a proposal for updates in XQuery has been presented in [TIHW01].

*XQuery.* XQuery *embeds* XPath syntax into SQL/OQL-style constructs. The keywords in XQuery clauses are `FOR - LET - WHERE - RETURN` (FLWR):

FOR *variable* IN *xpath-expr*
LET *additional_variable* := *xpath-expr*
WHERE *filters*
RETURN *xml-expr*

The `FOR` clause defines variables which are bound to the elements of iterating over the result set of XPath expressions. Additional variables may be defined in the `LET` clause. The result from the `FOR` and `LET` clauses is a sequence of tuples of variable bindings. The `WHERE` clause – similar to SQL/OQL – states additional conditions which tuples of variable bindings are used for generating the result. Then, the `RETURN` clause generates an XML subtree for each variable binding.

*Updates.* [TIHW01] describes a proposal for updates in XML, based on the XML Query Data Model. A language-independent definition of updates is given; the only assumption is that the updates are executed in an environment based on variable bindings (as, e.g., in XML-QL or XQuery). The following operations can be applied to an element node:

– Delete(*child*): if *child* is a member (i.e., a subelement (including `PCDATA`), an attribute, or a reference in an `IDREF(S)` attribute), it is removed.
– Rename(*child,name*): if *child* is a non-`PCDATA` member (i.e., a subelement or an attribute), it is given a new name. Here, [TIHW01] does not specify what happens if an attribute is renamed to a name that already exists.
– Insert(*content*): *content* can be `PCDATA`, an element, an attribute, or a reference which is added to an element. Insertion is critical due to the unique-parent-requirement in the XML Query Data Model; see below.
– Analogously for InsertBefore(*child,content*) and InsertAfter(*child,content*).
– Replace(*child,content*) is equivalent to InsertBefore(*child,content*) followed by Delete(*child*).

Here, *content* can either be a variable, or an XML pattern containing variables for generating new structures.

Based on the abstract operations, [TIHW01] proposes to extend XQuery by `FOR-LET-WHERE-UPDATE` clauses where the `UPDATE` clause is of one of the forms:

```
DELETE $child
RENAME $child TO name
INSERT [content] [BEFORE|AFTER $child]
REPLACE $child WITH [content]
```

### 3.3 Data Manipulation in the DOM/XML Query Data Model

Since [TIHW01] uses the XML Query Data Model, the constraints described in Section 2.1 apply. The *unique-parent* restriction constrains the in-place restructuring of a given XML tree. If any variable in $\boxed{content}$ is bound to a subtree, there are the following alternatives:

– Moving the subtree: then, the original subtree is moved, i.e., *re*moved from its original position, breaking the relationship described by this child-relationship. On the other hand, all references to elements in this tree come along.
– Deep-copying, using new `ID`s: this leads to the problems described below for generating new `ID`s and adapting `IDREF`s. Maintaining reference attributes between copied trees during complex restructuring and integration tasks becomes at least expensive, if not impossible at all. Moreover, references *into* the copied tree must be decided to point into the original or into the copy.

**Example 1 (Updates in the DOM model)** *Assume two trees $A$ and $R$(esult), where subtrees $A_1$ and $A_2$ that reference each other will be copied during the process to $R$.*

*First, $A_1$ is copied from $A$ to $A'_1$ in $R$, duplicating it. It is reasonable to adapt all references inside the copied fragment $A_1$ to $A'_1$. Since $A_2$ is not yet copied (even not known to be copied later), references from $A'_1$ to $A_2$ cannot be adapted; they still point to $A_2$ in $A$. Wrt. $R$ they are dangling. Then, some other operations are done.*

*Later, $A_2$ is copied to $A'_2$ somewhere in $R$, its internal references are adjusted.* Now, *references from $A'_1$ in $R$ to nodes in $A_2$ could be adjusted to $A'_2$:*

*"adjust all references in the target tree that point into the subtree which is copied."*

*This is not correct in the general case when a tree $R_1$ is copied inside $R$ to $R_2$ – here only the internal references should be adjusted. Refine the above with*

*"... if the subtree to be copied is not already part of the target tree."*

*Now, $A'_2$ in $R$ contains references to $A_1$ in $A$ – wrt. $R$, they are dangling now. How is it possible to adjust the references from $A_2$ to $A_1$ into references from $A'_2$ to $A'_1$? Note that $A_1$, $A_2$, and $A'_1$ may be changed in the meantime.*

The proposal for updating XML in [TIHW01] has to solve this problem.

## 4 Data Integration: Merge and Fusion Operations

Moreover, for data integration in general, not only "simple" updates are desired, but also specialized operations on tree fragments. The result is constructed using subtrees, elements, and literals of the input sources by the following operations:

– Linking and collecting: elements and tree fragments are linked together,
– Element fusion: identifying elements in different sources that represent the same object in the application domain,
– Synonyms: identifying and renaming properties,
– Projection: define tree views of the integrated database.

As illustrated below, the above operations require a data model that is more expressive and more flexible than the XML tree models described above.

**Fusing Elements and Subtrees.** *Fusing* elements that 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, and
2. collects the attributes and subelements of both original elements.

Here, (1) conflicts with the unique-parent relationship. For (2), it is preferable that the data is not copied, but the result uses the original data.

**Example 2 (Object Fusion)** *Consider two data sources as shown below and in Figure 1(a). Both describe countries, where* cia *contains information about name, area, population, and capital, and* gs *contains information about cities.*

```
<!ELEMENT cia (country+)>
<!ELEMENT country (border*)>
   <!ATTLIST country  name CDATA #REQUIRED      car_code ID #REQUIRED
                      area CDATA #IMPLIED       population CDATA #IMPLIED
                      capital CDATA #REQUIRED>
<!ELEMENT border (#PCDATA)>    <!ATTLIST border country IDREF #REQUIRED>
<!ELEMENT gs (country+)>
<!ELEMENT country (city+)>  <!ATTLIST country  name CDATA #REQUIRED>
<!ELEMENT city EMPTY>
   <!ATTLIST city name CDATA #REQUIRED pop CDATA #REQUIRED>
```

*Excerpts of the instances:*

```
<cia>
  <country car_code='D'  capital='Berlin'
           name='Germany'  area='356910'
           population='83536115'>
     <border country='F'>451</border>
     <border country='A'>784</border>
       :
  </country>
     :
</cia>
```

```
<gs>
  <country name='Germany'>
    <city name="Berlin" pop="3472009"/>
    <city name="Hamburg" pop="1705872"/>
      :
  </country>
    :
</gs>
```

*An obvious and typical integration step is to unify the* countries *in the* cia *tree with the* countries *in the* gs *tree. In XPathLog (Sec. 6), this is done by the rule*

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

*The example is continued below, Figure 1(b) depicts the final result.*

**Synonyms.** Names are also subject of operations, e.g., the integrated database uses a unified terminology that differs from the source terminologies. Often, it is not recommended to generate new relationships between nodes, but to introduce the target terminology as synonyms for already existing relationships, only defining a relationship between names (of subelements and attributes).

**Example 3 (Integration: Synonyms)**
*Especially, synonyms are an efficient means for taking a whole property from a source tree (and namespace) to the result tree: Consider the situation obtained in Example 2 where the following synonyms are defined:*

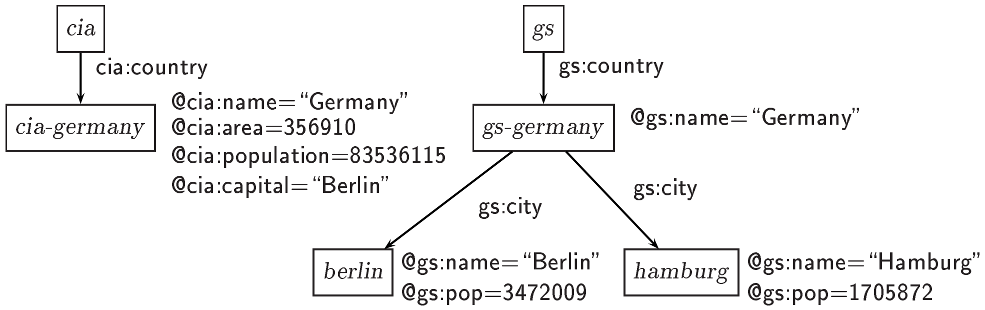
```
cia:name = name.                gs:city = city.          gs:text() = text().
cia:area = area.                gs:name = name.
cia:population = population.    gs:pop = population.
```

Synonyms can also be used for mediating between ontologies, mapping names to one or more target ontologies.

**Adding Links.** Additionally, the integrated database often contains additional links between elements that originally belong to different sources. These can either be represented by attributes, or by subelement relationships (contributing to a new result tree view).

**Example 4 (Integration: Additional Links)**
*The integration is completed by linking the country subtrees to a result tree and adding the* capital *reference attributes, here, using* germany[@cia:capital="Berlin"] *and* berlin[name="Berlin"]*. The resulting tree fragment is given in Figure 1(b). In XPathLog (cf. Section 6), this is done by the rules*

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




(a) Element fusion – before

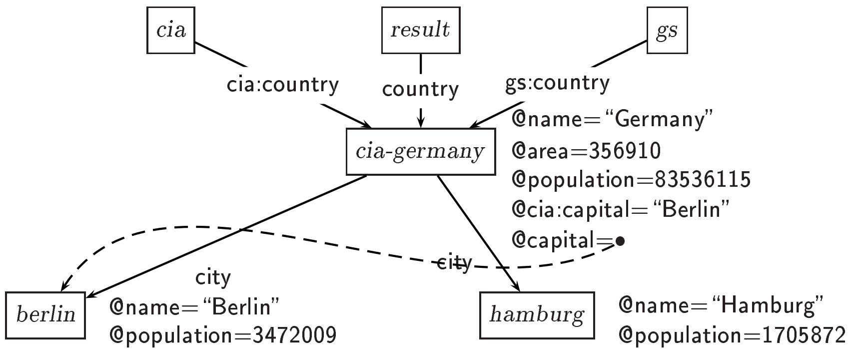



**Fig. 1.** (b) Element fusion – after

**Example 5** *Consider another restructuring of the database given in Example 2, extended with a database about organizations:*

<!ELEMENT orgs (organization+)>
<!ELEMENT organization (member_names*)>
<!ATTLIST organization abbrev ID #REQUIRED name CDATA #REQUIRED>
<!ELEMENT member_names (#PCDATA)>
<!ATTLIST member_names type CDATA #REQUIRED>

*Excerpt of the instance:*

<orgs> <organization abbrev='UN' name='United Nations'>
<member_names type='member'>Germany</member_names>
<member_names type='observer'>Switzerland</member_names>
:
</organization>
:
</orgs>

*The result trees (each of them represents a single organization) should satisfy the following DTD:*

<!ELEMENT organization (member*)> <!ATTLIST organization ...>
<!ELEMENT member (*the same as country from the combined cia/gs sources*)>
*Every country subtree becomes a* member *subelement of multiple organizations:*

O[member→C] :-
    orgs/organization→O[member_names→N], result/country→C[@name→N].

*Here, in a unique-parent model, it is necessary to generate (and maintain) multiple identical copies; note that the* country/border/@country *reference attributes must be accordingly adapted.*

The above operations for *element fusion*, *linking*, and *synonyms* allow for powerful integration concepts for generating a result database from a set of sources.

**Projection.** When the integration and restructuring process is completed, projections are used to define *result views* of the internal database. A result view is an XML tree, e.g., specified by a root node and a DTD.

**Example 6** *Consider again Example 5. Here, taking any* organization *node as the root of an* organization *tree, the* organizations *DTD defines the desired projection (pruning the* member_names *elements).*

In the following, we present an alternative data model which is tailored to the requirements of data integration from several source trees.

## 5 XTreeGraph: The Alternative Data Model

A crucial feature of a data integration language is that the tasks described above can be specified in an intuitive way, and that the language is powerful enough to allow for short and concise statements. The DOM and XML Query Data Model are not suitable for such operations due to the *unique parent* and *unique name* restrictions.

Instead, we propose to distinguish between the *internal* data model which is more powerful than the pure XML tree data models, and an export data model – which is the XML data model. The internal database is not intended to represent "the XML document", but a *database* representing a forest of *overlapping* XML trees: it is an edge-labeled graph (similar to the XML-QL data model), called *XTreeGraph*, that is especially tailored to the requirements of data integration. From this database, XML documents may be defined as *views* – some of the views are the original XML trees; subtrees can belong to several tree views. The XTreeGraph model extends the DOM idea. The main features of the model are:

- the universe contains the element nodes of the XML instance, literals used in attribute values and text contents, and element and attribute names, making them full citizens of the model;
- elements have properties: (i) subelements (ordered) and (ii) attributes (unordered), similar to the DOM and XML Query Data Model;
- multivalued attributes (`NMTOKENS` and `IDREFS`) are silently split;

– reference attributes (`IDREF` and `IDREFS`) are silently resolved.

Although it basically "looks like" DOM, the data model significantly differs from the DOM and XML Query Data Model: the data model does not impose any additional requirements on the child and parent relationships – thus,

– elements may have more than one parent, i.e., belong to several trees (or, *tree views*),
– the structure may even be cyclic (the external tree views to be defined have to care for finiteness),
– there is no global order of elements, but the children relationship is ordered for each individual element.

The XTreeGraph data model combines the features of the DOM with support for the additional features for data integration described in Section 4: the internal model is not a forest, but a graph that *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* that recursively consists of subelements and attributes (specified by a suitable projection to a result signature). The use of namespaces especially allows to identify the original trees even when the internal database has been changed into a densely connected graph.

The XTreeGraph is formally described by an *X-structure* (the full theory can be found in [May01c]):

**Definition 1 (X-Structure)** *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 that associates with every $x \in \mathcal{V}$ two lists of pairs, representing the child and attribute axes:*
  - $\mathcal{A}_{\mathcal{X}}(\mathsf{child}, x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})^{\mathbb{N}}$ *and*
  - $\mathcal{A}_{\mathcal{X}}(\mathsf{attribute}, x) \in ((\mathcal{V} \cup \mathcal{L}) \times \mathcal{N})^{\mathbb{N}}$ *as an arbitrary enumeration; recall that reference attributes are resolved silently into references to nodes and multivalued attributes are split.*

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 $\mathcal{A}_{\mathcal{X}}(\mathsf{child}, x)$ according to the XML specification.

Note that the attribute and element names in $\mathcal{N}$ are full citizens of the universe (which e.g. may occur in predicates and can be bound to variables).

### 5.1 Relationship with Standard XML Notions

As stated above, the XTreeGraph is an extension of the DOM and the XML Query Data Model, thus, all central notions such as the DOM API, XPath, XPointer, XML-QL, XQuery, XML Schema etc. also apply to it. Especially, in Section 6, we describe an XPath-based querying and manipulation language that has been defined and implemented based on this data model.

**DOM.** The elements are still "elements" in the sense of the XML data model and the DOM – i.e., they have children and attributes. Thus, an implementation of this data model can also provide the DOM API functionality.

**XPath.** XPath expressions are evaluated in the same way as for DOM. Only for the parent, ancestor, and sibling axis, the steps may result in steps in different trees – here, the range can be restricted by the *nodetest* (e.g., to a given namespace), or the expressions can be evaluated wrt. *result tree views*.

### 5.2 Operations

Below, we show how this model supports data integration by

- adding arbitrary subelement links for defining overlapping trees,
- equating elements that represent the same real-world entity in different sources ("*fusing*" objects); these can be made a single element in the internal database (and a subtree of both original trees), and
- equating *names* and *synonyms*.

**Accessing Sources.** When adding an XML document to the XTreeGraph, it is mapped to its canonical X-structure as a separate tree, accessible by a constant. Optionally, the tree is equipped with a namespace. 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 that 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 that describe 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).

**Generating a Result Tree by Selecting and Linking Subtrees.** The XTreeGraph is then extended by *linking* subtrees of the original documents. 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.

**Linking.** According to the syntax described in Section 3.2, linking is expressed by $e'.\mathsf{insert}(e)$ where $e$ is an already existing element. $e$ (i.e., the subtree rooted in $e$) can be made a subtree with any tag $t$ of some other element $e'$ by inserting the pair $(e, t)$ into $\mathcal{A}_\mathcal{X}(\mathsf{child}, e')$.

**Fusing Elements.** When *fusing* elements by $e_1 = e_2$, all occurrences of $e_2$ in the X-structure are replaced by (w.l.o.g.) $e_1$. Then,

1. the result is a subelement where one of the original elements was a subelement: every pair $(t', e_2) \in \mathcal{A}_\mathcal{X}(\mathsf{child}, e)$ is replaced by an entry $(t', e_1)$ at the same position,

2. $e_1$ collects the attributes of both original elements, i.e., $\mathcal{A}_\mathcal{X}(\mathsf{attribute}, e_2)$ is appended to $\mathcal{A}_\mathcal{X}(\mathsf{attribute}, e_1)$ (handling multi-valued attributes suitably),
3. $e_1$ collects the element and text contents of both original elements, i.e., the items of $\mathcal{A}_\mathcal{X}(\mathsf{child}, e_2)$ are appended/inserted to $\mathcal{A}_\mathcal{X}(\mathsf{child}, e_1)$ (where the ordering strategy of subelements must be specified e.g. by a result DTD).

**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. Then, the same navigation path in the graph can be used by different names: After equating $n_1 = n_2$, searching for $n_1$-subelements of an element $e$ results in a list containing all $e'$ s.t. $(n_i, e') \in \mathcal{A}_\mathcal{X}(\mathsf{child}, e)$ for $i = 1, 2$.

**Projection by Signature.** Projections in an XTreeGraph are specified by a root node and a namespace, or, more detailed, by a *signature* (e.g. given by a DTD), describing which subelements and attributes recursively belong to the view. Then, e.g., XPath expressions can be evaluated wrt. the result tree view.

**Summary.** Using the above strategies, the internal database can be seen as a "three-level" model:

1. the "basic" layer: source(s) provide tree structures that *may* be used.
2. the "internal" layer: the internal XTreeGraph is manipulated by integration operations.
3. the "export" layer: the result trees are then defined as *tree views* over the internal database.

### 5.3 Metadata: Advanced Integration Issues

In general, *metadata* is used for data integration, either merely based on schema information, e.g. in heterogeneous or distributed databases (projects such as TSIMMIS [GMPQ+97]), or also on *ontologies* that can be seen as databases on meta-data, giving hints how application-semantical concepts are related.

For representing metadata, XML provides the concepts of DTDs, XML Schema, and RDF. Expecially, since XML Schema documents are valid XML instances and the XML Schema definition is tailored to describe the XML Query Data Model, they can be added directly to the XML database. Then, integration can homogeneously use the data trees and the metadata trees.

*Ontologies.* Ontologies that are accessible in XML format can also be added as XML trees to the database in order to guide the integration process. Integration steps can the use (i) data, (ii) metadata like XML Schema, and (iii) additional ontology databases. Then results from reasoning on the meta-level 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 about cities in european countries, and a database about economics in the G7 countries) and use them for integrating databases,

also using analogy reasoning. Ontology-based and similarity-based XML data integration in this framework is described in [May01e].

## 6 A Realization: XPathLog and LoPiX

The above data model is implemented in the LoPiX system [May01b], using XPathLog [May01d] as a querying, data manipulation, and data integration language.

**XPathLog.** XPathLog extends XPath with the Datalog-style variable concept (and with implicit dereferencing). The variables are bound to the names, nodes, and literals that result from the respective match when evaluating the underlying XPath expression. The semantics of XPathLog queries wrt. an X-structure is derived from the formal semantics of XPath given in [Wad99]. XPathLog rules are logical rules over XPathLog expressions. Using *logical expressions* for specifying an update is perhaps the most important difference to approaches like XSLT, XML-QL, or XQuery where the structure to be *generated* is always specified by XML patterns. Existing nodes are communicated via variables to the head, where they are *modified* when appearing at host position of atoms. 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*). Atomic updates are then executed on $\mathcal{A}_{\mathcal{X}}(\mathsf{child}, e)$ and $\mathcal{A}_{\mathcal{X}}(\mathsf{attribute}, e)$. Details can be found in [May01d,May01c].

*Expressiveness and Complexity.* The relationship between XPath, XML-QL, XQuery, and the XML Query Algebra on one side, and and XPathLog on the other is similar as between SQL and the relational algebra on one side, and Datalog on the other side: The output size of XML-QL and XQuery statements is always polynomially bounded wrt. the input size (and also the computational complexity is polynomial). In contrast, XPathLog allows for recursion – similar to Datalog. Thus, problems like e.g. the transitive closure can be expressed. Even more, the result trees can be infinite (infinitely iterating the generation of new elements), i.e., the execution of XPathLog programs can be nonterminating (same as Datalog with function symbols or object creation, and as F-Logic). Note that XSLT – as a functional-style transformation language – has the same expressive power as XPathLog. Also XML-QL and XQuery can be used for generating new XML documents from given ones since their output format is XML, but they can only compute polynomial transformations.

**The LoPiX System.** LoPiX has been developed using major components from the Florid system [LHL+98], an implementation (in C++) of F-Logic [KLW95]. Due to the similarities between the F-Logic data model and the XML data model in general, and the XTreeGraphs in particular, the Florid modules provided a solid base for an XPathLog implementation. Especially the functionality of the complete module for the evaluation of a deductive language over a data model with complex objects could be reused. The system architecture of LoPiX is depicted in Figure 2.

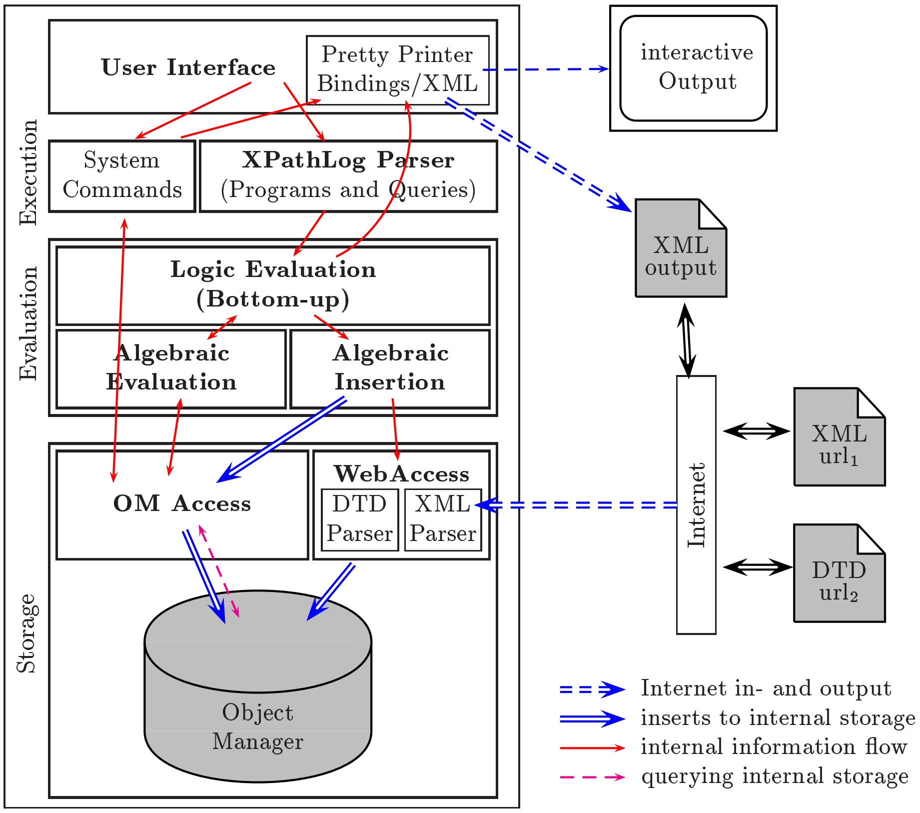



**Fig. 2.** Architecture of the LoPiX System

The actual (extensional) database, i.e., the X-structure is stored in the *ObjectManager*. The *ObjectManagerAccess* implements the abstract XTreeGraph data model based on the contents of the *ObjectManager*. Here, the intensional properties include derived axes, transitivity of class hierarchy, object fusion, synonyms, built-in functionality for data conversion etc. The *WebAccess* functionality is closely intertwined with the *OMAccess* module: XML sources are mapped to trees in the internal database. The *OMAccess* is accessed declaratively from the *Evaluation* component (*LogicEvaluation, AlgebraicEvaluation*, and *AlgebraicInsert*) that provides in fact a *generic* implementation of a deductive language over a data model with complex objects.

## 7 Related Work and Conclusion

As already mentioned in the introduction, pre-XML approaches to object-oriented and semi-structured data, e.g., Good [GPBG94], OEM [GMPQ+97], Strudel [FFK+98], YATL [CDSS99], and Florid [LHL+98] used graph-based "proprietary" semi-structured data models. With these, also Datalog style languages have been used for manipulating and integrating semi-structured data(bases).

The OEM-based LOREL language has been migrated to XML in [GMW99]; YATL has been adapted to XML in [CCS00].

Several querying and transformation languages have already been mentioned in the introduction; most of them are based on XPath. XSLT is a pure transformation language, thus the above problems do not occur. XML-QL [DFF+99] is a querying and transformation language based on matching of XML-style patterns; note that XML-QL uses the graph-based STRUDEL data model. Although, the language has not yet been extended with update constructs. XML Data integration in an XML-QL-based environment is described in [BGL+99]. XQuery is a comprehensive XPath-based XML query language that also directly produces XML output. [TIHW01] propose updates in XQuery (cf. Section 3.2); an implementation is not yet available, and the described problems are not addressed in detail. There are also several XML database-like systems, e.g., TAMINO [Sof] and eXcelon [eXc], using proprietary internal data models based on the DOM. Several approaches have been presented that implement XML over (object-)relational database management systems (e.g., SILKROUTE [FTS00], XPERANTO [CFI+00], and STORED [DFS00]). Since their *abstract* data model is still the XML tree, the above operations are also not supported. The same holds for the algorithms presented in [TIHW01].

*Conclusion.* By using an *edge-labeled* graph instead of the *node-labeled* tree in the XML data model, and refraining from the "unique-parent" constraint of the DOM and XML Query Data Model, the restrictions of the DOM and XML Query Data Model can be overcome. The presented data model is a compatible *extension* to the DOM and XML Query Data Model. On the other hand, it supports the presented operations for data integration. The data model is the base of the LoPiX system [May01b] that implements XPathLog as a declarative XML data integration language.

## References


[Beh01] E. Behrends. Einbindung und Evaluierung einer DOM-Speicherungskomponente in LoPiX. Diploma Thesis, Universität Freiburg (in german), 2001.

[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.

[CCS00] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 141–152, 2000.

[CDSS99] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators need Data Conversion. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999.

[CFI+00] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPeranto: Publishing object-relational data as XML. In *WebDB 2000*, 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. W3C Technical Report, `www.w3.org/TR/NOTE-xml-ql`.

[DFS00] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 2000.

[eXc] eXcelon Corp. XML Application Development. `http://www.exceloncorp.com/`.

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1998.

[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.

[FTS00] M. Fernandez, W.-C. Tan, and D. Suciu. Silk Route: Trading between Relations and XML. In *World Wide Web Conference (WWW 9)*, 2000.

[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. *Journal of Intelligent Information Systems*, 8(2), 1997.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. *WebDB 1999*, 1999.

[GPBG94] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[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.

[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. LoPiX: A System for XML Data Integration and Manipulation. In *Intl. Conf. on Very Large Data Bases (VLDB), Demonstration Track*, 2001.

[May01c] W. May. XPath-Logic and XPathLog: A Logic-Based Approach for Declarative XML Data Manipulation. Technical report, Universität Freiburg, Institut für Informatik, 2001. See `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[May01d] W. May. A Rule-based Querying and Updating Language for XML. In *Workshop on Databases and Programming Languages (DBPL 2001)*.

[May01e] W. May. A Framework for Generic Integration of XML Sources. In *Intl. Workshop on Knowledge Representation meets Databases (KRDB 2001)*.

[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)*, Springer LNCS 1727, 1999.

[Sof] Software AG. Tamino. `http://www.tamino.com/`.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Halevy, and D. Weld. Updating XML. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2001.

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

[W3C] W3C – The World Wide Web Consortium `http://www.w3.org`.

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