A Framework for Generic Integration of XML Sources

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

Abstract

We consider the situation where several XML sources have to be integrated which are assumed to contain complementary, overlapping contents. These overlappings have to be detected, and then appropriate operations have to be applied to the internal database to generate a result view. The approach uses the XPathLog language for formulating queries and updates of an XML database.

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. In contrast to classical data integration, due to the world-wide accessibility of XML sources, the ad-hoc integration of autonomous sources became important.

We use XPathLog [May01d], a Datalog-style extension of XPath [XPa99], for integration of autonomous XML sources from the Web. In contrast to other approaches, an extended XPath syntax and semantics is also used for a declarative specification how the database should be *updated*. Due to the close relationship with XPath, the semantics of rules is easy to grasp. In our approach, XML instances are mapped to an internal graph model [MB01] which allows for special operations tailored to information integration. XPathLog is implemented in the LoPiX system [LoP01]. The practicability of XPathLog and LoPiX for *integration programs* in the style of "explicit" classical rule-based programs which are tailored to the given sources has been shown in [May01c].

In this paper, we show how the approach extends to a generic integration of sources, using ontologies and applying heuristics for identifying overlapping contents. Starting with the original XML trees, the internal database is developed into a graph database which represents multiple, overlapping XML trees as the integrated union of the sources. Also there can be multiple *result tree views* which are used for answering user queries. This requires a data model which is different from the pure XML tree model: trees have to be *combined* by *merging* elements, *linking* subtrees, and *identifying* properties by defining synonyms.

The paper is structured as follows: we continue with a short description of the system architecture and introduce a running example. Section 2 describes the XPathLog data manipulation language. The focus of the paper follows then in Section 3, describing the integration operations and applying them in the context of heuristics-based data integration, i.e., searching for items in the database which are likely to have some correspondence to each other. Section 4 concludes the paper.

LoPiX. The LoPiX system [LoP01] implements XPathLog over an internal graph-based data model, called *XTreeGraph*, and adds data-driven Web access functionality. LoPiX consists of a central XPathLog engine, a storage module which implements the XTreeGraph data model, an Internet access module, and an optional XML/XPath interface for electronic data interchange with external XML-based systems.

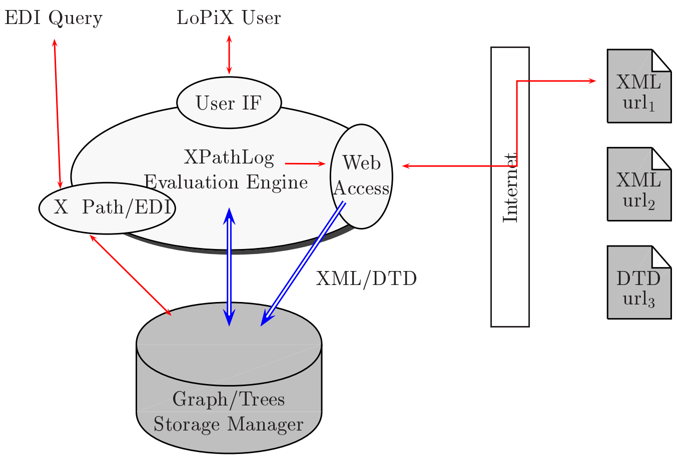

Figure 1: LoPiX architecture

LoPiX follows the *warehouse* approach for data integration: all sources are loaded into the internal database before integrating them (in contrast to the *virtual* approach where the sources are accessed only on-demand). Based on the copied sources, an integrated graph database is computed by restructuring and adding links, i.e., without further copying. The integrated views are then defined as tree views over the internal database. For heuristics-based integration, the *warehouse* approach is preferable, since the heuristics have to be applied to the *complete* information, even if the actual query only needs a small excerpt of the view which is then defined based on the heuristics. Often it is sufficient to use only “typical” excerpts of the sources for applying heuristics to define a global mapping which is then used to access the sources again for the actual query – thus, the *integration* task may be separated from the actual querying. In this case, a combined warehouse and virtual strategy can be applied.

Example. Consider the following situation: when buying a digital camera, the catalogs of the producers (which include technical information) and catalogs of different retailers are considered. E.g. an excerpt of the *Nikon* producer tree is as follows:

```
<producer name="Nikon">
  <product type="digital camera" name="Coolpix880" mpix="3.34" price="1799.00">
    <zoom> <external focallengthlow="8" focallengthhigh="20">
             <digital factor="4"> </zoom>
    <accessory type="lens" name="WC-E24"/>
                  <!-- accessory/@name is an IDREF attribute -->
```

```
      <accessory type="lens" name="TC-E2"/>
      <accessory type="lens" name="TC-E3"/>
    </product>
    <product type="wide angle adapter" name="WC-E24" factor="0.66" price="219.00">
    </product>
    <product type="teleconverter" name="TC-E2" factor="2" price="259.00">
    </product>
    <product type="teleconverter" name="TC-E3" factor="3" price="589.00">
    </product>
    :
  </producer>
```

In contrast, a retailer tree maybe looks as follows:

```
  <store name="shop1">
      <digitalcamera producer="Nikon" type="Coolpix880" price="1699.00"/>
      <digitalcamera producer="Nikon" type="Coolpix990" price="2399.00"/>
      <digitalaccessory producer="Nikon" type="WC-E24" price="199.00"/>
      <digitalaccessory producer="Nikon" type="TC-E2" price="269.00"/>
      <digitalcamera producer="Olympus" type="C3000" price="1599.00"/>
      :
  </store>
```

Each of the retailers sells products of different brands; although they often do not offer all products. So there are overlapping and non-overlapping portions of contents. The overlappings are used for integration across the sources, then providing also an integrated view of the non-overlapping parts.

2 XPathLog: The Data Manipulation Language

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]).

XPathLog as an XML query language. As an addressing mechanism, XPath provides the base for most XML querying languages, which extend it with special constructs. For XPathLog [May01d], the extension feature are Datalog style variables, joins, and rules. Variables can be introduced in XPath expressions as follows:

- by *nodetest*→X or *nodetest*[*filter*]→X, binding the variable to the element node or attribute node or text node which is "traversed" at this point when evaluating the underlying XPath expression,
- at the *nodeTest* position; in this case, the variable is bound to the element name or attribute name. This allows for binding variables to metadata notions.

Additionally, XPathLog allows for implicitly dereferencing of `IDREF` attributes, e.g., //a/@b/c. For the formal semantics, see [May01a].

Example 1 (XPathLog) *The following expression returns all tuples (N, P, F, AP) such that there is a digital camera model with name N and price P, which can be combined with a teleconverter with factor F and price AP.*

```
?- //product[@type→"digital camera" and @name→N and @price→P]
        /accessory[type="lens" and
              @name[@type→"teleconverter" and @factor→F and @price→AP]].
N/"Coolpix880"  P/1799.00  F/2  AP/259.00
N/"Coolpix880"  P/1799.00  F/3  AP/589.00
```

Data Model: XTreeGraph. The approach uses internally the *XTreeGraph* data model [MB01]. It is based on an *edge-labeled navigation graph* which is a variant of the semistructured data model defined in [Bun97]. In contrast to the XML Query Data Model, the XTreeGraph data model is especially tailored to the requirements of data integration (note that the querying fragment of XPathLog does not depend on a special data model, it also applies to the DOM or the XML Query Data Model):

- nodes may have multiple parents. Thus, the *internal database* is not only an XML tree, but represents an *XML database* containing multiple, possibly overlapping trees. This modeling allows for *linking* subtrees into other trees, and for *fusing* elements which then accumulate all properties of the original elements.
- the labels (i.e., the element and attribute names which are used for navigation) are elements of the universe, allowing for variables ranging over names, and supporting powerful operations on names, e.g., *synonymizing* them for defining additional access navigation paths (without introducing additional links).
- namespaces are supported; every source can be equipped with an own namespace.
- support for *signature information* which can e.g., be extracted from a DTD. Result trees may be defined as views according to a given signature.

XPathLog as an XML update language. XPathLog rules serve for a declarative specification of database updates. If the body of a clause evaluates to true for some assignment of its variables, the stored XML data is updated such that the head also evaluates to true: When used in the head, the / operator and the [...] construct specify which properties should be added or updated.

In the head, [...] does not act as a filter, but as a *constructor*: e.g., the following rule adds the producer's url as an attribute to the above producer tree (and assigns the constant nikon to the *Nikon* company tree):

```
T[@url→"www.nikon.com"], T = nikon :- //producer→T[@name→"Nikon"].
```

New elements can either be created as *free* elements by atoms of the form /name[...] or as subelements of the form C[name→E]. The following rule creates new product subelements to the *Nikon* tree with some properties:

```
nikon/product[@type→"fisheye adapter" and @name→"FC-E8" and @price→"569.00"].
nikon/product[@type→"digital camera" and @name→"Coolpix950" and @price→".."].
```

The fisheye adapter is added as an accessory lens to the *Coolpix880* camera, generating a subelement with an `IDREF` attribute:

```
P/accessory[@type→"lens" and @name→A] :-nikon/product[@name="Coolpix880"]→P,
                                          nikon/product[@name="FC-E8"]→A.
```

Elements can be *linked* as subelements to others, creating *overlapping* (sub)trees, e.g.,

```
P/accessory→A :-nikon/product[@name="Coolpix950"]→P,
                 nikon/product[@name="Coolpix880" and accessory→A].
```

makes all accessory subelements of the *Coolpix880* subtree to be also accessory subelements of the *Coolpix950* (note that the elements are still stored only once, but the XTreeGraph contains two subelement edges to each of them).

For details, formal semantics, and more examples, see [May01a]. 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. In the following section, we show how the update functionality is used for generating an integrated XTreeGraph database from several sources.

3 Heuristics-Based Data Integration

The internal data model supports data-intensive integration tasks by implementing as many operations as possible in a view-like style, fusing and re-linking instead of copying subtrees and introducing (global) synonyms instead of lots of additional links. The use of XPathLog as a database programming language for writing mediator-like specialized *integration programs* for *known* sources is described in [May01c]; its applicability has been demonstrated in the case study [May01b].

In this paper, we continue the approach one step further: the schemas of the sources are not known for writing the integration program, but coincident concepts and overlapping contents have to be detected by the program. Data-driven and heuristics-driven integration is often supported by ontologies which describe the application domain and can be used to lead the integration process. Additionally, strategies can exploit structural similarities and overlapping ranges in the source trees. While already the integration of known sources profits much from a clear and concise declarative language, the expressive power – especially for stating rules which combine the handling of data and metadata – is an indispensible requirement for integration of unexplored sources. In the following, we assume that the XTreeGraph also contains a (very simplistic) ontology tree, associated with the constant photo_ontology:

```
<ontology name="photography">
  <concept name="product">                  <!-- concept/@name: ID attribute -->
    <property name="name" value="string"
    <property name="price" value="amount" unit="currency"/>
  </concept>
```

```
  <concept name="accessory" isa="product"/>   <!-- concept/@isa: IDREFS attribute -->
  <concept name="camera" isa="product"/>
  <concept name="digitalcamera" isa="camera">
    <property name="mpix" value="amount"/>
    <property name="zoom"> <!-- a complex property --> </property>
  </concept>
  <concept name="lens" isa="accessory">
    <property name="factor" value="amount"/>
  </concept>
  <concept name="teleconverter" isa="lens"/>
  <concept name="fisheye" isa="lens"/>
</ontology>
```

Transitivity of the isa relationship is e.g. specified by the XPathLog rule

```
C[@isa→C2] :- photo_ontology//concept→C[@isa/@isa→C2].
```

Note that the ontology is not directly connected with the DTD or XMLSchema definition of any of the sources, but it is useful if the ontology covers the semantic notions of the sources. The integration also profits if the names used in the sources and the concepts of the ontology coincide. It is even possible to add knowledge to the ontology tree, e.g., concept names in foreign languages.

3.1 Operations

XPathLog and the LoPiX system provide the following features and operations for data integration (in the following, we assume that a suitable ontology tree is also available, rooted in the constant ontology):

Namespaces. Parsed source trees can be equipped with namespaces, e.g., nikon: and shop1: above. Variables can also be bound to namespaces.

Element Fusion. Elements of different source trees which are identified as equivalent can be *merged*, making the result a child of both parents. In this case, the elements are *fused*, e.g., the result accumulates all attributes and subelements of the original elements. When source-namespaces are used, the sources of the individual properties can still be inferred.
E.g., the rule (employing variables also at namespace and property position)

```
X = Y :-//S1:producer[@S1:name→PN]/S1:product[@S1:name→N]→X,    % producer
         //S2:store/S2:PType[@S2:producer→PN and @S2:type→N]→Y,  % retailer
         photo_ontology/concept[@name=PType and @isa/@name→"product"].
```

merges elements in a producer tree (namespace bound to S1) with appropriate elements in retailer trees (S2). Note that PType occurs as a *navigation* variable; XPathLog silently maps between strings and constants when used at property position (e.g., PType binds to "digital camera" which is mapped to the name constant digitalcamera).
Applying the above rule creates e.g. a merged element for *Coolpix880*:

```
<nikon:product nikon:type="digital camera" nikon:name="Coolpix880" nikon:mpix="3.34"
   nikon:price="1799.00" shop1:type="Coolpix880" shop1:price="1699.00">
   <nikon:zoom> <nikon:external nikon:focallengthlow="8" nikon:focallengthhigh="20">
               <nikon:digital factor="4"> </nikon:zoom>
   <nikon:accessory @nikon:type="lens" nikon:name="WC-E24"/>
   <nikon:accessory @nikon:type="lens" nikon:name="TC-E2"/>
   <nikon:accessory @nikon:type="lens" nikon:name="TC-E3"/>
</nikon:product>
```

which collects all properties, i.e., attributes and subelements from the original elements

nikon/product[@name="Coolpix880"] and
//store[@name="shop1"]/digitalcamera[@type="Coolpix880"].

This new element is both a nikon:product subelement of the nikon: tree, *and* it is a shop1:digitalcamera subelement of the shop1: tree.

Synonyms. Synonyms may be used for defining "result" properties from properties of the original trees, e.g.,

```
NS:X =product_category :- E[@NS:X→PType],
      photo_ontology/concept[@name=PType and @isa/@name→"product"].
```

identifies the *name* nikon:type with product_category as a synonym (not adding any new link in the graph database). Synonyms allow for overlaying sources which use different names for identical concepts with a unified terminology.

Linking. The result tree view is collected by suitably *linking* subtrees of the original sources (which possibly use different element names) to the result tree, without removing them from the original tree. The use of an *edge-labeled* graph allows for flexibility in the naming of properties: When linking an element, the "name" (i.e., tag) under which the element occurs in the new tree may differ from its original element name:

```
result[camera→C] :- //producer/product[@product_category="digital camera"]→C.
```

using the above synonym product_category for e.g., the nikon:type attribute of products. Then, e.g., the query

```
?- result/camera[@name→N and @S:price→P].
N/"Coolpix880" S/nikon P/1799.00
N/"Coolpix880" S/shop1 P/1699.00
N/"Coolpix950" S/nikon P/2599.00
N/"Coolpix950" S/shop1 P/2399.00
```

uses the result root node in combination with the information from the original trees. Using this functionality, it is possible to define several (overlapping) "view" trees on the original documents.

3.2 Strategies

Using the above operations for fusing and linking objects, and for introducing synonyms, the expressive power of the language (especially, variables at namespace and property name positions) can be exploited for a declarative implementation of strategies for heuristics-based data integration.

Ontology-based integration. In addition to the original data sources, metadata such as ontologies and dictionaries can be used to relate concepts of different sources; here variables at property position allow for an intuitive and declarative specification of integration rules. The integration process is then based on detecting overlappings, i.e., objects and properties which are described in different source trees. Many approaches to data integration use "ontology-based" strategies, e.g., in this case,

- for objects (i.e., elements): if their element names are detected to be in some sense "equivalent" by using the ontology, and if they coincide by their "key values" (according to the ontology information). Here, rules extending the following pattern can be used:

```
X = Y :- ontology/concept[@name→M and @key→K1 and @equivalent→C2],
         //M→X[K1→V1],  //N→Y[K2→V2],  C2[@name→N and @key→K2].
```

- for properties: properties defined in different sources are merged if they apply to concepts which are regarded to be equivalent, and the properties are equivalent wrt. the ontology.

  The straightforward approach is to define a derived property of all objects of the host concept based on the original properties, e.g., (the signature atom M[Q⇒C] declares Q as a desired result)

```
X[Q →V] :- //M→X[NS:P→V], M[Q⇒_],    <-- using a signature atom -->
           ontology/concept[@name→M and property[@name = P and @equivalent→Q]].
```

  By using *synonyms*, properties can be handled globally without adding links in the internal database:

```
NS:P = Q :- //M→X[NS:P→V], M[Q⇒_],
            ontology/concept[@name→M and property[@name = P and @equivalent→Q]].
```

The above strategy depends on a "complete" ontology, and it does not validate whether the actions are correct.

Analogy-based integration. Additionally, structural analogies in the database can (i) be used to check if two objects are actually equivalent, and (ii) to find additional equivalences which are not covered by the ontology. Both aspects are based on graph-theoretic investigations by isolating fragments (= views) of the database which are assumed to be equivalent:

- detect potentially equivalent objects,
- detect equivalent properties and remove properties from the view which do not match,
- try to extend the equivalent fragments with further properties or nodes.

Example 2 *From the Nikon reference source, the names of products are known. Then, in a retailer's tree, a set of elements is identified which has a property whose range coincides or overlaps (if the retailer does not sell all Nikon products, and perhaps also Olympus and Minolta products). The remaining properties of these elements are then compared with the known properties, and a pattern is derived how to map the reference products to the retailers offers. This mapping can then be generalized to the Minolta and Olympus products which makes these also "full members" of the integrated* *database: e.g., their technical data can be compared with the Nikon products.*

Here, the rules are closely related to *deep-equality* in object-oriented databases. In [MLL97], it has been shown that such rules are expressible in an intuitive, short, and concise way in a language which allows for variables at property positions. Thus, *generic* rules can be given which detect overlapping fragments. Note that derived properties may also be considered (e.g., combining navigation steps).

If a valid overlapping is found, the next step is to materialize and evaluate it: not only corresponding objects are found, but also corresponding *concepts* are found which correlate concepts of different sources and thus generalize also to the non-overlapping part and serve for homogeneizing the data. By iterating object fusion, introducing synonyms, and finding overlappings, a homogeneous integrated database using the "preferred" concepts of the target ontology is constructed.

Finally, a result tree view is defined by adding appropriate subelement links and defining a result signature. The result tree then serves for answering queries (e.g., how to combine a bundle for a digital camera which satisfies the user's requirements, and to choose from which retailer the individual components are ordered).

4 Conclusion

Related Work. The XPathLog language has been designed as a crossbreed between F-Logic [KLW95] and XPath [XPa99] (also, LoPiX has been developed based on the Florid system), extending and applying the experiences with F-Logic and Florid for integration of semistructured data [LHL+98].

XML-QL [DFF+99] is another XML query language which is also based on a graph-based model. Thus, an extension with update operators potentially allows for the implementation of similar strategies. XML data integration in an XML-QL-based environment is described in [BGL+99].

Since XQuery [XQu01] uses the tree-based DOM/XML Query Data Model, it is not possible for an element to have several parents which severely restricts the update functionality. Especially, re-linking and fusing elements is not possible. A proposal for updates in XQuery has been presented in [TIHW01], but no solution for the above problem has been presented. The approach is not yet implemented.

Conclusion. We have presented an approach to XML data integration by using the XTreeGraph data model and the XPathLog language. Following the F-Logic tradition, XPathLog allows for powerful and declarative rules for implementing integration techniques for XML data. The close relationship with XPath ensures that its declarative semantics is well understood from the XML perspective. The graph-based data model supports operations such as linking and merging elements, and introducing synonyms which are crucial for declarative data integration.

We have sketched the integration of data sources where the schema is not previously known at programming time by detecting content overlappings based on ontologies and applying heuristics. The approach extends to the case where the source trees are not preselected, but, e.g., given as an answer to some query against a Web indexing service.

References

[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.

[Bun97] P. Buneman. Semistructured Data (invited tutorial). *ACM Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona, 1997.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. *8th. WWW Conference*. W3C, 1999. World Wide Web Consortium Technical Report, NOTE-xml-ql-19980819, `www.w3.org/TR/NOTE-xml-ql`.

[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, 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.

[LoP01] W. May. LoPiX: A System for XML Data Integration and Manipulation. *Intl. Conf. on Very Large Data Bases (VLDB), Demonstration Track*, 2001. See also `http://www.informatik.uni-freiburg.de/~may/lopix`.

[May01a] W. May. A Logic-Based Approach for Declarative XML Data Manipulation. Technical report, Universität Freiburg, Institut für Informatik, 2001. Available from `http://www.informatik.uni-freiburg.de/~may/lopix/`.

[May01b] 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`.

[May01c] W. May. Integration of XML Data in XPathLog. *CAiSE Workshop "Data Integration over the Web" (DIWeb'01)*, 2001.

[May01d] W. May. XPathLog: A Declarative, Native XML Data Manipulation Language. *International Database Engineering and Applications Workshop (IDEAS'01)*. IEEE Computer Science Press, 2001.

[MB01] W. May and E. Behrends. On an XML Data Model for Data Integration. *Intl. Workshop on Foundations of Models and Languages for Data and Objects (FMLDO 2001)*, Springer LNCS, 2001.

[MLL97] W. May, B. Ludäscher, and G. Lausen. Well-Founded Semantics for Deductive Object-Oriented Database Languages. *Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, Springer LNCS 1341, 1997.

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