From Semistructured Data to XML: Migrating the Lore Data Model and Query Language[*]

Roy Goldman, Jason McHugh, Jennifer Widom
Stanford University
{royg,mchughj,widom}@db.stanford.edu, http://www-db.stanford.edu/lore

Research on *semistructured data* over the last several years has focused on data models, query languages, and systems where the database is modeled as some form of labeled, directed graph [Abi97, Bun97]. The recent emergence of *eXtensible Markup Language* (*XML*) as a new standard for data representation and exchange on the World-Wide Web has drawn significant attention [BPSM98]. Researchers have casually observed a striking similarity between semistructured data models and XML. While similarities do abound, some key differences dictate changes to any existing data model, query language, or DBMS for semistructured data in order to fully support XML. This paper describes our experiences migrating the *Lore* database management system for semistructured data [MAG+97] to work with XML. We present our modified data model, whose definition was a subtly challenging task given that XML itself is just a textual language. Based on this model, we describe changes to *Lorel*, Lore's query language. We also briefly discuss changes to Lore's dynamic structural summaries (*DataGuides* [GW97]) and the relationship of DataGuides to XML's *Document Type Definitions* (*DTDs*).

1 OEM and Lorel

Lore is a complete database management system designed specifically to handle semistructured data [MAG+97]. Lore's original data model, *OEM* (for *Object Exchange Model*), is a simple, self-describing, nested object model that can intuitively be thought of as a labeled, directed graph [PGMW95]. In OEM all entities are *objects* that can either be *atomic* or *complex*. Each object has a unique *object identifier* (*oid*). Atomic objects contain a value from one of the atomic types, e.g., `integer`, `real`, `string`, `gif`, etc.. A complex object's value is a set of $\langle$*label*, *subobject*$\rangle$ pairs, where each label gives a textual description of the relationships between the object and its subobject. In the graph view of an OEM database, objects are nodes in the graph, complex objects have outgoing edges labeled with the relationship to their subobjects, and atomic objects contain their value. A single object in OEM may serve multiple roles if it has multiple incoming edges, possibly with different labels. A *Name* is an alias for a single object and serves as an entry point into the database graph. Lore's query language, *Lorel*, has a familiar `select-from-where` syntax and is based on *OQL* [Cat94], with certain modifications and extensions that are useful when querying semistructured data. Details on Lorel can be found in [AQM+97].

2 XML

XML is a textual language quickly gaining popularity for data representation and exchange on the Web [BPSM98]. Nested, tagged elements are the building blocks of XML. Each tagged element has a sequence of zero or more attribute/value pairs, and a sequence of zero or more subelements. These subelements may themselves be tagged elements, or they may be "tagless" segments of text data. Because XML was defined as a textual language rather than a data model, an XML document always has implicit order—order that may or may not be relevant but is nonetheless unavoidable in a textual representation. A *well-formed* XML document places no restrictions on tags, attribute names, or nesting patterns. Alternatively, a document can be accompanied by a *Document Type Definition (DTD)*, essentially a grammar for restricting the tags and structure of a document. An XML document satisfying a DTD grammar is considered *valid*. While not exactly a data model, a standard *Document Object Model* (*DOM*) for XML has been defined [AB+98], to enable XML to be manipulated by software. The DOM defines how to translate an XML document into data structures and thus can serve as a starting point for any XML data model.

In addition to attributes in XML, and the fact that XML is ordered and OEM is not, another obvious difference between XML and OEM is the treatment of tags/labels. In OEM, labels are used only as entry points and to denote relationships to other objects—an OEM object need not have a single label that it "owns". In contrast, the XML DOM specifies that each (non-text) element contains its identifying tag. Another key difference is that the XML DOM today does not directly support graph structure (as opposed to trees), no doubt an artifact of XML's document orientation. Currently, XML uses

[*]This work was supported by Rome Laboratories under Air Force Contract F30602-96-1-0312, by the National Science Foundation under grant IIS-9811947, and by a Schlumberger graduate research fellowship.

special attribute types to encode graph structure. An element can have a single attribute of type *ID* whose value provides a unique identifier that can be referenced by attributes of type *IDREF* or *IDREFS* from other elements. Consider this simple example:

```
<Person Id='P1' Name='Jeff Ullman' Colleague='P2'/>
<Person Id='P2' Name='Jennifer Widom' Colleague='P1'/>
<Publication Title='A First Course in Database Systems' Author='P1 P2'/>
```

Assume attribute `Id` is of type ID, `Colleague` is of type IDREF, and `Author` is of type IDREFS. The above example encodes a graph where the `Colleague` and `Author` attributes serve as labeled references to `Person` elements (similar to labeled subobjects in OEM).[1]

XML's "second-class" support of graph structure leads to interesting decisions in specifying a true data model and query language. Should an XML data model be a tree that corresponds to XML's text representation (like the DOM), or a graph that includes the intended links? Our view is that both approaches are important. In some situations, an application may wish to process XML data as a literal tree, where IDREF(S) attributes are nothing more than text strings. In other situations, an application may wish to process XML data as its intended semantic graph. Our decision is to support both modes—*literal* and *semantic*—which a user or application can select between. The choice of mode has a direct impact on query evaluation and results, as we will see later.

3 Lore's XML Data Model

In Lore's new XML-based data model, an XML element is a pair $\langle eid, value \rangle$, where *eid* is a unique *element identifier*, and *value* is either an atomic text string or a complex value containing the following four components:

1. A string-valued *tag* corresponding to the XML tag for that element.

2. An ordered list of *attribute-name/atomic-value pairs*, where each attribute-name is a string and each atomic-value has an atomic type[2] drawn from `integer`, `real`, `string`, etc., or `ID`, `IDREF`, or `IDREFS`.

3. An ordered list of *crosslink subelements* of the form $\langle label, eid \rangle$, where *label* is a string. Crosslink subelements are introduced via an attribute of type `IDREF` or `IDREFS`.

4. An ordered list of *normal subelements* of the form $\langle label, eid \rangle$, where *label* is a string. Normal subelements are introduced via lexical nesting within an XML document.

We differentiate normal subelements (4) from crosslink subelements (3) so we can support both literal and semantic modes in our model.

An XML document is mapped easily into our data model. Note that we ignore comments and whitespace between tagged elements. As a base case, text between tags is translated into an atomic text element; we do the same thing for `CDATA` sections, used in XML to escape text that might otherwise be interpreted as markup [BPSM98]. Otherwise, a document element is translated into a complex data element such that:

1. The tag of the data element is the tag of the document element.

2. The list of attribute-name/atomic-value pairs in the data element is derived directly from the document element's attribute list.

3. For each attribute value i of type IDREF in the document element, or component i of an attribute value of type IDREFS, there is one crosslink subelement $\langle label, eid \rangle$ in the data element, where *label* is the corresponding attribute name and *eid* identifies the unique data element whose *ID* attribute value matches i.

4. The subelements of the document element appear, in order, as the normal subelements of the data element. The label for each data subelement is the tag of that document subelement, or `Text` if the document subelement is atomic.

Note that multiple XML documents can be loaded into a single database, and any system of cross-document links (e.g., *XLink* or *XPointer*) can be used provided information that uniquely identifies elements is not lost.

[1]Unfortunately, currently in XML a DTD is required to specify attribute types, so it is common to use inelegant heuristics to deduce ID/IDREF/IDREFS types when a DTD is not available.

[2]While the XML specification does not include attribute types, some extensions to XML do, and we have chosen to include attributes types in our model.

```
<DBGroup>
  <Member Name="Smith" Advisor="m1" >
    <Age>28</Age>
  </Member>
  <Member ID="m1" Project="p1">
    <Name>Jones</Name>
    <Advisor>Ullman</Advisor>
  </Member>
  <Project ID="p1" Member="m1">
    <Title>Lore</Title>
  </Project>
</DBGroup>
```

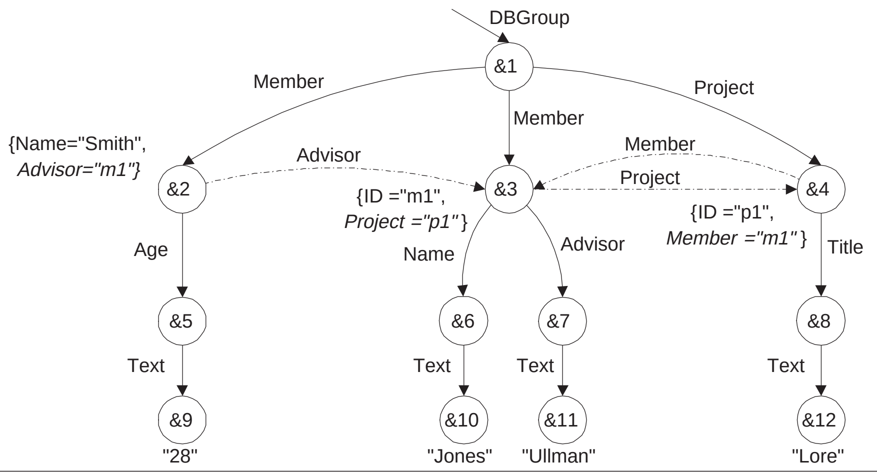

Figure 1: An XML document and its graph

Once one or more XML documents are mapped into our data model it is convenient to visualize the data as a directed, labeled, ordered graph. The nodes in the graph represent the data elements and the edges represent the element-subelement relationship. Each node representing a complex data element contains a tag and an ordered list of attribute-name/atomic-value pairs; atomic data element nodes contain string values. There are two different types of edges in the graph: (i) normal subelement edges, labeled with the tag of the destination subelement; (ii) crosslink edges, labeled with the attribute name that introduced the crosslink. Note that the graph representation is isomorphic to the data model, so they can be discussed interchangeably.

As mentioned earlier, it is useful to view the XML data in one of two modes: *semantic* or *literal*. Semantic mode is used when the user or application wishes to view the database as an interconnected graph. The graph representing the semantic mode omits attributes of type IDREF and IDREFS, and the distinction between subelement and crosslink edges is gone. Literal mode is available when the user wishes to view the database as an XML document. IDREF and IDREFS attributes are visible as textual strings, while crosslink edges are invisible. In literal mode, the database is always a tree.

Figure 1 shows a small sample XML document and the graph representation in our data model. Element identifiers (eids) appear within nodes and are written as &1, &2, etc. Attribute-name/atomic-value pairs are shown next to the associated nodes (surrounded by {}), with IDREF attributes in italics. Subelement edges are solid and crosslink edges are dashed. The ordering of subelements is left-to-right. We have not shown the tag associated with each element since it is straightforward to deduce for this simple database. (For example, node &3 has the tag *Member* and not *Advisor*.) In semantic mode, the database in Figure 1 does not include the (italicized) IDREF attributes. In literal mode, the (dashed) crosslinks are not included. Note that there is some structural heterogeneity in the data even though the sample data was kept purposefully small.

4 Lore's XML Query Language

We now discuss the modifications we have made to the Lorel query language to accommodate the differences between our new XML data model and OEM, and to exploit XML features not present in OEM. Recall that a database in our data model can be interpreted either in *semantic mode* or in *literal mode*. For simplicity let us assume that the desired mode is selected for each query.

Distinguishing between attributes and subelements. *Path expressions* are the basic building blocks of the Lorel language [AQM$^+$97]. A path expression in Lorel is essentially a sequence of labels, e.g., `DBGroup.Member.Project`, which may include label wildcards and regular expression operators. During query evaluation, path expressions are matched to paths in the database graph. For XML, we extend the meaning of path expressions to navigate both attributes and subelements, and we introduce *path expression qualifiers* in order to distinguish between the two when desired. We use the optional symbol `>` before a label to indicate matching subelements only, and the optional symbol `@` to indicate matching attributes only. When no qualifier is given, both attributes and subelements are matched—we expect this to be the most common case. Table 1 shows simple examples of path expressions with qualifiers applied over the database in Figure 1.

Qualification	Symbol	Example	Matches in semantic mode	Matches in literal mode
Subelements only	>	DB.Member.>Name	&6	&6
		DB.Member.>Advisor	&3, &7	&7
Attributes only	@	DB.Member.@Name	“Smith”	“Smith”
		DB.Member.@Advisor	*empty*	“m1”
None	None	DB.Member.Advisor	&3, &7	&7, “m1”

Table 1: Path expression qualifiers

Function	Description
Flatten(e)	Ignoring all tags, recursively serializes all text values in the subtree rooted at element e (following normal subelements only).
Concatenate(e)	Concatenates all immediate text children of element e and ignores all other subelements.
Tag(e)	Returns the XML tag of element e.
Eid(e)	Returns a string representation of the eid of element e.
XML(e)	Transforms the graph, starting with element e, into an XML document. Note that there is no single “correct” way to generate an XML document from graph-structured data, so it will be difficult to use this option to compare against string constants.

Table 2: Functions to transform elements into strings

Recall from Section 3 that in semantic mode IDREF(S) attributes are not visible, while in literal mode IDREF(S) are treated like other attributes and crosslink edges are not visible.

Comparisons. We anticipate that many different kinds of comparisons may be useful in queries over XML data. For example, constants might be compared against attribute values or against element text. We might want to compare against a serialization of all text elements in an XML subtree, ignoring markup. In graph-structured data, we might want to test for eid equality. Rather than supporting many distinct comparison operators, we decided instead that for the purpose of comparisons we would treat each XML component as some kind of atomic value, either through default behavior or via explicit transformation functions, as follows.

Attribute values are always atomic. For elements, Table 2 describes several built-in functions that can be used to transform an element into a string; these functions can be used outside of comparisons if desired, e.g., in the `select` clause. (Each function returns NULL if called over an attribute instead of an element.) Since it is inconvenient for a user to have to specify functions for every comparison, keeping in the spirit of Lorel we set default semantics when functions are not supplied based on our impression of the most common and intuitive uses:

1. For an atomic (`Text`) element, the default value is the text itself.
2. For elements that have no attributes and only one or more `Text` elements as children, the default value is the concatenation of the children’s text values (a restricted case of the *Concatenate* function).
3. For all other elements, the default value is the element’s eid represented as a string (the *Eid* function).

Example: Suppose we are looking for group members whose advisor is “Ullman”. In the original version of Lorel, `DBGroup.Member.Advisor = "Ullman"` does the trick. Based on Figure 1 it appears that for our XML data model we must write `DBGroup.Member.Advisor.Text = "Ullman"`, and indeed this expression will give us the correct answer. However, note that the former comparison also will give us the correct answer by virtue of default semantics case (2) above. Space limitations preclude numerous examples in this paper, but in general we have found that most Lorel queries designed for an OEM database can be used unmodified on a corresponding XML database, such as the simple example we have just shown.

Range qualifiers. We have extended Lorel so that the expression “`[`*range*`]`” can optionally be applied to any path expression component or variable. The *range* is a list of single numbers and/or ranges, e.g., `[2-4,7]`. When such a *range qualifier* is applied to a label in a path expression, we limit matched values to those within the range. For example, “`select y from DBGroup.Member x, x.Office[1-2] y`” returns the first two `Office` subelements of every group member.[3] When a range qualifier is applied to a variable then we limit the entire set of variable bindings to the specified range. For example, "`select y[1-2] from DBGroup.Member x, x.Office y`" returns the first two `Office` elements over all of the members in the database.

Order-by clause. The result of a query is an ordered list of eids identifying the elements selected by the query. (Any attributes in the query result are coerced into elements.) If there is no `order-by` clause in the query then the ordering is unspecified. In some applications it may be important for the query result to be ordered based on the original XML document. We term this ordering the *document order* of the database, and we extend the functionality of the standard Lorel `order-by` clause with an "`order-by document-order`" expression. (Newly constructed elements in query results, since they do not come directly from the original document, currently are placed at the end of the document order with an unspecified order among them.) Document order is frequently exactly what we want, but it can produce unintuitive results for graph-structured data. We have defined other orderings that seem appropriate in certain cases, but space limitations preclude their discussion in this paper.

Transformations and structured results. Using queries to restructure XML data may be more common than it was in OEM, so we have introduced two new query language constructs to transform data and return structured query results. The first construct, the `with` clause, is added to the standard `select-from-where` query form and was introduced originally in Lorel's view specification language. When a `with` clause is present in a query, the query result replicates all data selected by the `select` clause, along with all data reachable via any of a set of path expressions in the `with` clause. A complete description is given in [AGM+97].

We also have extended Lorel to support *Skolem functions* [End92], for more expressive data restructuring than was provided in Lorel previously. In Lorel, a Skolem function accepts an (optional) list of variables as arguments and produces one unique element for every binding of elements and/or attributes to the argument(s). When a new set of bindings for the arguments is passed into a Skolem function then a new database element is created and returned. Subsequent calls to the same function with the same argument bindings returns the same result element. Skolem functions are not new to semistructured query languages, first appearing in MSL [PAGM96] then later in StruQL [FFLS97] and YATL [CDSS98], but they are new to the XML version of Lorel.

Updates. Unlike most other semistructured and object-oriented query languages, Lorel supports an expressive, declarative *update language* [AQM+97]. Space limitations preclude discussing in detail the effect of XML on Lorel's update language, or more generally the intricacies of XML and updates, but let us touch upon the topic briefly. A number of changes discussed above carry over directly to our update language. Additional modifications include the ability to create both attributes and elements, and order-relevant updates (e.g., inserting after the fourth subelement). Note that it is not always obvious how semantic versus literal mode should be interpreted in the context of updates. A further issue is that of maintaining a correspondence between an XML database and an XML document in the face of updates. This issue has several interesting aspects to it, including ambiguity when serializing newly created database elements, and ensuring continued validity with respect to a document's DTD.

5 DataGuides and DTDs

Since semistructured databases generally have no predefined, fixed schema, we have introduced *DataGuides*—concise and accurate structural summaries of the underlying database [GW97]. A DataGuide is itself a graph, where every label path in the database appears exactly once in the DataGuide and every label path in the DataGuide exists in the original database.

When a DTD is not supplied, the notion of a DataGuide is just as important for XML as for OEM. We can reuse our original construction and maintenance algorithms for DataGuides with relatively minor modifications to accommodate our extended XML data model. One interesting aspect is the introduction of order. We can extend our DataGuide definition to incorporate order, but if we do it strictly then we may end up with very large DataGuides where much of the information is due to variations in order. Given that the goal of the DataGuide is to summarize the database, we are investigating techniques for using weighted averages over all instance data to determine order within the DataGuide.

When DTDs are used to restrict the data, then of course the DataGuide becomes less important. However, because DataGuides serve several functions in Lore and can be expensive to build, we can now build a DataGuide (actually, an *Approximate DataGuide* [GW99]) directly from a DTD. An interesting direction is to combine DataGuides with DTDs: it is easy to envision a scenario where DTDs are available for specific portions of an XML database, but the overall database

[3] We have decided that within each element we order attributes first, then crosslink subelements, then normal subelements. This ordering is relevant in cases where a label in a path expression may match more than one kind of XML component.

is still semistructured. We can build a DataGuide over the portions not governed by DTDs, with appropriate links to DTDs where appropriate. Note also that DTDs currently do not support graph structure beyond restricting attribute types to ID and IDREF(S), so DataGuides are more expressive than DTDs in this regard.

6 Conclusion and Status

We have described the fundamental changes we made in migrating Lore's data model and query language to support XML. We have nearly finished converting our Lore implementation to match the specifications in this document. We intend to look at performance enhancements to our Lore-XML system based on changes to the data model and query language, specifically in the areas of indexing and data layout. So far we have focused on using Lore as a fine-grained XML database rather than a coarse-grained document repository. We are also considering a new approach that mixes the decomposition of (portions of) XML documents into their components in the database, together with storing document copies and maintaining mappings between the two.

Acknowledgements

We are grateful to all the members of the Lore group and beyond who suffered through our interminable meetings about converting Lore to XML, particularly Vineet Gossain, Michael Rys, Anthony Tomasic, Frank Tompa, and Vasilis Vassalos for lively discussions. We also thank Vineet Gossain, Beverly Yang, and Stephen Patel for their implementation work.

References

[AB$^+$98] Editors: V. Apparao, S. Byrne, et al. Document object model (DOM) level 1 specification, October 1998. W3C Recommendation available at http://www.w3.org/TR/REC-DOM-Level-1.

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, Delphi, Greece, January 1997.

[AGM$^+$97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for semistructured data. In *Proceedings of the Workshop on Management of Semistructured Data*, pages 83–90, Tucson, Arizona, May 1997.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1):68–88, April 1997.

[BPSM98] Editors: T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible markup language (XML) 1.0, February 1998. W3C Recommendation available at http://www.w3.org/TR/1998/REC-xml-19980210.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 117–121, Tucson, Arizona, May 1997. Tutorial.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Francisco, California, 1994.

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 177–188, Seattle, Washington, June 1998.

[End92] H.B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, San Diego, California, 1992.

[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, September 1997.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the Twenty-Third International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[GW99] R. Goldman and J. Widom. Approximate DataGuides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel, January 1999.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, pages 413–424, Bombay, India, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.