XMorph: A Shape-Polymorphic, Domain-Specific XML Data Transformation Language

Curtis Dyreson[#1], Sourav Bhowmick[*2], Aswani Rao Jannu[#3], Kirankanth Mallampalli[#4], Shuohao Zhang[^5]

[#] *Department of Computer Science, Utah State University*
Logan, UT USA

[1] `Curtis.Dyreson@usu.edu`
[3] `aswani.jannu@usu.edu`
[4] `kirankanth.mallampalli@usu.edu`

[*] *Nanyang Technological University*
Singapore

[2] `assourav@ntu.edu.sg`

[^] *Marvel*
San Jose, CA USA

[5] `shuohao@msn.com`

Abstract— **By imposing a single hierarchy on data, XML makes queries *brittle* in the sense that a query might fail to produce the desired result if it is executed on the same data organized in a different hierarchy, or if the hierarchy evolves during the lifetime of an application. This paper presents a new transformation language, called XMorph, which supports more flexible querying. XMorph is a shape polymorphic language, that is, a single XMorph query can extract and transform data from differently-shaped hierarchies. The XMorph data shredder distills XML data into a graph of *closest relationships*, which are exploited by the query evaluation engine to produce a result in the shape specified by an XMorph query.**

I. INTRODUCTION

The goal of the research presented in this paper is to make it easier for users to query data, in particular XML data. One factor that adds complexity to querying XML data is that query writers have to know the *shape* of the data to effectively query it. Long before the advent of XML E. F. Codd wrote about this problem. In his foundational paper on the relational model Codd critiqued the hierarchical model, in part, because it uses *asymmetric path expressions* to locate data [4]. A path expression is a specification of a path in a hierarchy. Codd presented five hierarchies for a simple part/supplier database and demonstrated that, in general, a path expression formulated with respect to one hierarchy would fail on some other. For instance, suppose that the expression `supplier/part` locates parts "below" suppliers. The same expression fails when the data is organized differently, say when parts are above suppliers. Asymmetric path expressions have resurfaced in XML query languages.

In this paper we propose a new, shape-polymorphic, domain-specific data transformation language called XMorph. We invite readers to visit the XMorph project website[1] to experiment with XMorph in an on-line demo or download the Java implementation. XMorph offers the following features in a data transformation language.

Easy to specify and transform the data's shape. The primary component of XMorph is a *morph* in which the user declares the desired shape of the result. XMorph reorganizes the source data to match the specified shape.

Shape polymorphism. In XMorph, only the shape of the output needs to be given, the query adapts to the shape of the input. *Shape polymorphism* was first described by Jay and Crockett [7]. In shape polymorphism in object-oriented languages, a method, e.g., to print a value, adapts to the shape of the data, e.g., adapts to a tree or a list. This notion applies to database query languages as follows: a language is *shape polymorphic* if a query evaluated on the *same* data in *different* structures yields (approximately) the *same* result[2].

Ability to identify information loss. The XMorph query engine can analyze a query to determine potential information loss in a transformation.

XQuery support. XMorph can be translated to XQuery.

Ability to treat attributes as indistinct from sub-elements. Though data modelers often arbitrarily choose to use attributes rather than subelements, XMorph queries do not force users to differentiate between them.

Easy creation of groups. XQuery 1.0 has ad-hoc support for groups using a distinct-values function. XQuery 1.1 adds support for grouping in aggregation. XMorph supports both persistent and dynamic group creation for data transformation.

Vocabulary translation. To use XMorph, a user has to know the "vocabulary" (e.g., the names of the elements) in a data collection. But XMorph also supports vocabulary translation, so that users can change their terminology.

Finally, XMorph is a *domain-specific*[3] language, lacking many features found in a general-purpose query language like XQuery, such as namespace and white-space handling. XMorph can not guarantee that document order is maintained by a transformation (due to grouping, though without

[1] `http://www.cs.usu.edu/~cdyreson/pub/XMorph`

[2] The same result *modulo* duplicates, ordering, and attribute/sub-element swaps.

[3] Domain-specific has nothing to do with a database "domain," rather it means "special purpose" or dedicated to a specific task.

grouping, order can be maintained). Further XMorph assumes that the data has a meaningful vocabulary (i.e., element and attribute names), XMorph is not an appropriate tool for data collections in which the names are not meaningful (e.g., every element is named "foo"). We also assume that the vocabulary is small, relative to the data size. Finally, XMorph can give counterintuitive results when transforming recursive data since each level of recursion moves data that is semantically close to a node further from it.

II. Related Work

Previous shape-related research on making it easier to query XML can be broadly classified into four categories.

1) **Query relaxation/approximation.** Techniques have been proposed to find data that inexactly or approximately matches a query [1],[8] by relaxing the notion that only crisp answers be returned by query evaluation [3]. These techniques implicitly generate and explore a space of shapes that are related to the shape of the input and/or query, usually all shapes within a given edit distance. But these techniques are orthogonal to XMorph (they could be employed to create approximate XMorph).

2) **Query correction/refinement**. In this approach, similar queries are automatically generated when a query is unable to be satisfied [3] or a query is refined [2]. The user guides the search for the query they want to execute by choosing among alternatives, and these alternatives implicitly involve trying different shapes for the input. These approaches are also orthogonal to XMorph, which does not require interactive user input.

3) **XML search engines**. XML search engines have simple, easy-to-use interfaces (c.f., [5],[12]). Like XMorph, they de-couple queries from specific hierarchies. But unlike XMorph, XML search engine queries typically do not transform data. XMorph straddles the middle ground between XML search engines and path expression-dependent XML query languages by borrowing useful techniques from each end of the spectrum.

4) **Structure-independent querying**. The final category of research is more clearly applicable to XMorph. The idea that path expressions can be made (more) symmetric by exploiting a least common ancestor (LCA) has been explored previously. Schema-free XQuery uses the meaningful LCA [10], XSeek exploits node interconnections [5], and others use the smallest LCA [11],[12],[14]. Similarly, we proposed a closest XPath axis [15],[16] based on the LCA. In contrast to all of the above research, XMorph focuses on the use of the LCA in data trans-formation, characterizing the potential information loss, and explicitly specifying and mutating the shape of data.

Similar to XMorph there are other proposals for declarative languages for specifying transformations of XML [9],[13]. These languages hide from users many of specification details that would be needed in a language such as XQuery or XSLT. However, these techniques, unlike XMorph, are not shape polymorphic. A transformation query might have to be rewritten for a different hierarchy.

III. XMorph Overview

This section gives a short tutorial on XMorph through a series of examples of increasing complexity. The examples will transform the data about books written by E. F. Codd shown in Fig. 1.

The primary function in XMorph is a *morph*, which places children below a parent in the result. The parameter of the function is a *pattern*, which specifies the shape of the result. Fig. 2 gives a simple example. The query is intended to list the titles written by each author extracted from a collection of book data. The pattern specifies that `<title>` and `<name>` elements become children of `<author>`s[4]. Only `<title>` and `<name>` elements that are *closest* to an `<author>` element are placed within that `<author>`. The notion of *closeness* forms the core semantics for XMorph [15]. Intuitively the idea is that authors are closely related to the titles of their own books and articles (and their own names), but not close to titles written by others. Fig. 2 also shows the result of the query when evaluated on the data in Fig. 1.

A morph can be restricted to select individual authors. Suppose we want only the titles by the author E. F. Codd. Then we can use the query given in Fig. 3 which selects `<author>` elements *where* the value is '`E.F. Codd`'. The result is the same as that in Fig. 2 since only E. F. Codd has authored books in the source data. There may be duplicate authors in the data, but authors can be *grouped* to eliminate the duplicates. Fig. 4 shows an example that consolidates titles under a single E. F. Codd author using a 'group' modifier for the author.

Modifiers are listed after a label, separated by commas. The group modifier uses the default, persistent grouping for author (e.g., author is grouped by its 'key' as specified by the data's schema, or by the distinct-values function for a schema-less data collection). Authors could also be dynamically grouped during query evaluation, by specifying a grouping pattern.

E. F. Codd wrote both books and articles, and we may want to select only book titles. In the query given in Fig. 5, `<title>` elements closest to a `<book>` element are selected but books are *hidden* in the output; a `<book>` is only used to find a closely-related `<title>`. The result is the same as that in Fig. 2 since the data has only `<book>`s.

Though XMorph assumes that a user is familiar with the vocabulary of the data, it also has a *translate* function to automate translation of a query or its result into the terms desired by a user. The translation can be specified before a morph, with the output of the translation being piped into the morph (as shown in Fig. 6) or afterwards, in which case the output of the morph is piped into a *translate* function.

The *pipe* operator, '|', is used to connect the output of one XMorph function into the input of another function. Initially, the input is assumed to come from a *default* data collection, but the data collection could be explicitly named using a *data* specification as illustrated in Fig. 7.

[4] In the explanation of this example, we've assumed elements rather than attributes, but, in general, "author", "name", or "title" could be either an attribute or element.

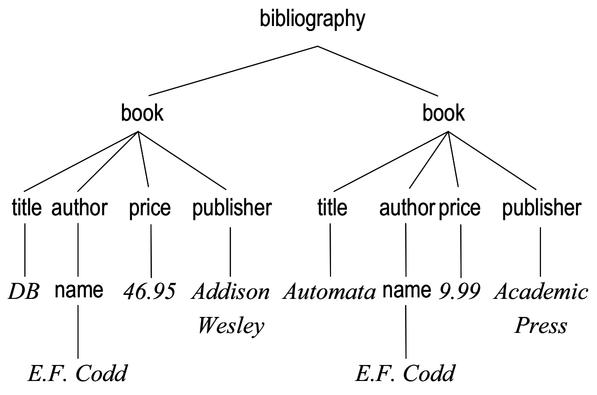

Fig. 1 Authors listed by book

```
morph author [
        name
        title
      ]
```

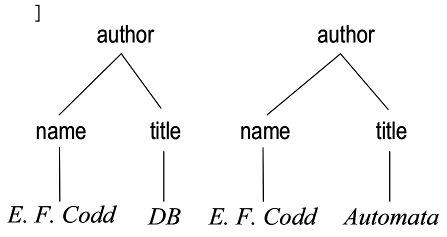

Fig. 2 List titles by author, and result

```
morph author [
        name, where value = 'E.F. Codd'
        title
      ]
```

Fig. 3 List titles by author E. F. Codd

```
morph author, group [
        name, where value = 'E.F. Codd'
        title
     ]
```

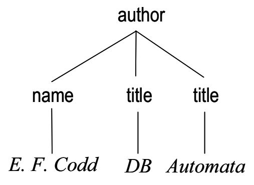

Fig. 4 List titles grouped by author E. F. Codd

To this point, the descriptions of the queries have avoided describing the shape of the input, that is, the same query could be applied to data in a variety of hierarchies. Moreover each query, when applied to the same data in different hierarchies will produce the same output. The only hierarchy that the user

```
morph author, group [
        name, where value = 'E.F. Codd'
        title [ book, hide ]
      ]
```

Fig. 5 List book titles grouped by E. F. Codd

```
translate
  author -> writer
| morph
    writer [
      name, where value = 'E.F. Codd'
      title [ book, hide ]
    ]
```

Fig. 6 List book titles by the writer E. F. Codd

needs to specify is that of the output. To illustrate this, consider the query shown in Fig. 8. The query applies a *morph* to the result of a *morph*. The first *morph* produces a hierarchy which lists the titles published in each year and within each title the authors for that title. The second morph is the transformation of Fig. 2.

XMorph also supports *mutation* of a shape. A mutation is similar to a morph but unlike a morph the entire shape is implicitly involved rather than just the portion explicitly given in the pattern. A mutation is given in Fig. 9. The mutation explicitly lists only three types, but it outputs the entire shape of the data, with two mutations. First it moves `<publisher>` elements to within the closest `<author>` elements, and second, it clones `<title>` elements to also place them under the closest `<author>` (the un-cloned `<title>` elements will remain in place). The rest of the shape is not changed.

These examples show a few of the uses of XMorph and illustrate its most important design feature: shape polymorphism. In a shape-polymorphic query language a user specifies the shape of the output. The evaluation of a query adapts to the shape of the input data to produce the desired output. In XMorph, this adaptation is based on the notion of closeness. While XQuery path expressions are wedded to a particular hierarchy, the key to developing a technique that works for any hierarchy is to identify what is *invariant* across the "same" data organized in different hierarchies. Observe the two hierarchies in Fig. 1 and Fig. 10. In each of the hierarchies, the book titles by an author are *closest* to that author. Here "closeness" is roughly defined as the distance on the path between nodes in the hierarchical model of an XML document. This is not something specific to authors and titles only. In fact, whenever two nodes are closest in Fig. 1 so are their counterparts in Fig. 10.

XMorph can be evaluated natively or translated to XQuery. As an example, Fig. 11 shows the XQuery translation of the query in Fig. 2 on the data of Fig. 10. The translation uses the shape of the data to generate the path expressions to locate closest elements.

```
data 'dblp.xml'
  | morph author [ name [ title ] ]
```

Fig. 7 List titles by author from dblp.xml

```
morph year, group [
          title [ author [ name ] ]
        ]
| morph author [ name title ]
```

Fig. 8 Morphing a morph

```
mutate author [
          publisher
          title, clone
        ]
```

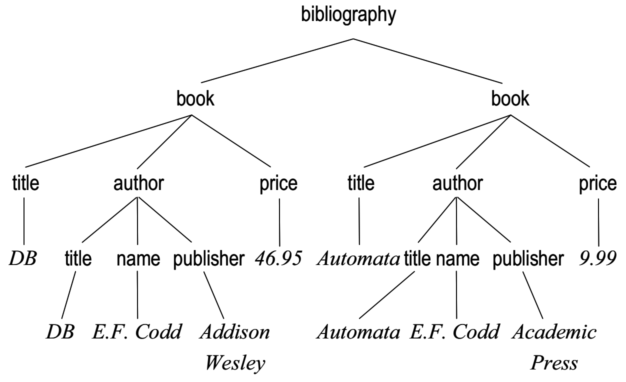

Fig. 9 Mutating the data

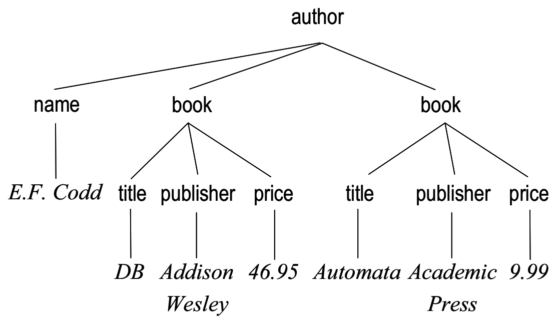

Fig. 10 Books listed by author

```
for $a in /author, $n in $a/name,
    $t in $a/book/title
return <author>{$a/text()}
          {$n}{$t}
        </author>
```

Fig. 11 XQuery for the query of Fig. 2 on the data of Fig. 10

IV. SUMMARY

XQuery is precise but brittle. An XQuery programmer can use path expressions that precisely locate data. But a programmer has to be familiar with the shape of the data to effectively query it. And if that shape changes, or if the shape is other than what the programmer expects, then the query may fail. An XML search engine is easy to use but imprecise. Not much is required of search engine users, but XML search engine queries dispense with the shape of data entirely. Between these two extremes are shape polymorphic query languages. Queries in such languages avoid both the brittleness of XQuery and the loss of shape in XML search engine queries. This paper presents XMorph, a shape polymorphic data transformation language for XML. An XMorph query uses, mutates, and extends the data's shape into a shape desired by the user. The constructed shape is subsequently used to render the data.

In future we plan to implement XMorph on a back-end SQL database with XMorph rendering to SQL. We would also like to apply XMorph in data integration; differently-shaped data can be translated to a common representation to improve the comparison and integration of data.

ACKNOWLEDGEMENTS

This research was supported by a Utah State University New Faculty Research Grant.

REFERENCES

[1] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree Pattern Relaxation," in Proceedings of EDBT, 2002, pp. 89-102.
[2] A. Balmin, L. Colby, E. Curtmola, Q. Li, and F. Ozcan, "Search Driven Analysis of Heterogeneous XML Data," in CIDR, 2009.
[3] T. Brodianskiy and S. Cohen, "Self-Correcting Queries in XML," in Proceedings of CIKM, 2007, pp. 11-20.
[4] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. CACM 13(6): 377-387 (1970).
[5] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv, "XSEarch: A Semantic Search Engine for XML," in Proceedings of VLDB, Berlin, Germany, 2003, pp. 45-56.
[6] C. Dyreson and S. Zhang, "The Benefits of Utilizing Closeness in XML," in DEXA Workshops, 2008, pp. 269-273.
[7] C. B. Jay and J. R. B. Crockett, "Shapely types and shape polymorphism," in European Sym. on Programming, 1994, pp. 302-316.
[8] Y. Kanza, W. Nutt, Y. Sagiv, "Flexible Queries over Semistructured Data," in PODS, June 2001, pp. 40-51.
[9] S. Krishnamurthi, K. Gray, and P. Graunke, "Transformation-by-example for XML," in Workshop of Practical Aspects of Declarative Languages, LNCS 1753, 2000, pp. 249-262.
[10] Y. Li, C. Yu, and H. V. Jagadish. "Schema-Free XQuery," Proceedings of VLDB, Sep. 2004, Toronto, CA, pp. 72-83.
[11] Z. Liu, J. Walker, and Y. Chen. "XSeek: a semantic XML search engine using keywords," in VLDB, 2007, pp. 1330-1333.
[12] Z. Liu and Y. Chen, "Identifying Meaningful Return Information for XML Keyword Search," in *SIGMOD*, 2007, pp. 329-340.
[13] T. Pankowski. "A High-Level Language for Specifying XML Data Transformations," in Proceedings of ADBIS, LNCS 3255, 2004.
[14] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases," in SIGMOD, 2005, pp. 527-538.
[15] S. Zhang and C. Dyreson, "Symmetrically Exploiting XML," in WWW, 2006 Edinburgh, Scotland, May 2006, pp. 103-111.
[16] S. Zhang and C. Dyreson, "Polymorphic XML Restructuring," in IIWeb (a WWW Workshop), 2006. iiweb2006/cs.uiuc/edu/6.pdf.