In: *XML 2000*

XML Queries and Transformations for End Users

Martin Erwig
Oregon State University
Department of Computer Science
Corvallis, OR 97331, USA
erwig@cs.orst.edu

Abstract

We propose a form-based interface to expresses XML queries and transformations by so-called "document patterns" that describe properties of the requested information and optionally specify how the found results should be reformatted or restructured. The interface is targeted at casual users who want a fast and easy way to find information in XML data resources. By using dynamic forms an intuitive and easy-to-use interface is obtained that can be used to solve a wide spectrum of tasks, ranging from simple selections and projections to advanced data restructuring tasks. The interface is especially suited for end users since it can be used without having to learn a programming or query language and without knowing anything about (query or XML) language syntax, DTDs or schemas. Nevertheless, DTDs can be well exploited, in particular, on the user interface level to support the semi-automatic construction of queries.

1 Introduction

XML is an emerging standard for data exchange. Moreover, it is very likely that the vast amount of tomorrow's data and web resources are available in XML format. Thus, there is a very strong demand for languages to process XML documents and data resources. In particular, querying and transforming XML data from one format into another will be a frequent task. The fact that almost every computer user is, via the Internet, a potential user of XML data presents a challenge to design powerful yet easy-to-use languages for processing XML. In general, it cannot be expected that end users will be able or willing to learn sophisticated XML query languages. In particular, one cannot even assume that end users have any knowledge about XML, not to speak of DTD, XSLT, and so on. On the other hand, search engines that are solely based on key words are not powerful enough to exploit the structure that the XML format contributes to data. Experience with relational databases has shown that form-based query and update facilities rather than, say, SQL, are accepted and employed by end users.

Thus, there is a very strong need for a user-friendly query interfaces to XML documents. The impact of designing and implementing such interfaces is very high since end users make up such a huge group of XML clients. This can well be the largest group of XML user at all, and maybe even among the largest group of users of any software package.

In the following we will present an approach to query and transform XML data that is particularly aimed at end users. The general idea is to let users draw samples of documents they are interested in. By using a visualization of XML that mimics forms people are used to deal with we achieve an easy and direct way of expressing XML queries. In particular, this approach does not require users to learn a query language. The query mechanism scales up (at least to some degree), that is, users can start with very simple keyword-like searches and can advance by adding structural requirements. In addition, reformatting and restructuring of query results is also possible.

In the next section we demonstrate by examples how textual query languages can be currently used to process XML. From this we derive design goals for an end user query system. Our proposal for such a query interface is based on an intuitive visualization of XML as nested boxes, called the *document metaphor*, which we describe in Section 3. In Section 4 we show how elementary queries can be posed by simply drawing example documents, called *document patterns*, that should match the sought information. More advanced queries, and in particular, XML transformations, can be expressed by pairs of document patterns, called *document rules*: the first pattern essentially specifies the selection of data, whereas the second pattern can be used to express projections and restructuring of data. This is shown in Section 5. All the XML querying and processing is possible without presence of DTDs, in particular, users do not have to be aware of DTDs at all. In Section 6 we demonstrate that DTDs can still be very nicely utilized to obtain menu-driven interfaces to construct queries. Related work is discussed in Section 7, and finally, Section 8 presents some conclusions.

2 XML Query Languages

There exist many different data model and query language proposals that can be used to query and transform XML data. With a few exceptions all these are textual query languages that have either evolved from database query languages or that have emerged from XML itself. To give an impression of these and to illustrate why we believe they are *not* suited for end users we consider an example application and show how queries can be expressed in already existing languages.

Assume we have a document (or database) providing information about music resources, such as CDs, scores, MIDI-files, lyrics, video-clips, and so on. An small excerpt is shown below.

```
<music>
  <cd year="1993">
    <group>Spyro Gyra</group>
    <title>Dreams Beyond Control</title>
    <track no="1">
      <title>Walk the Walk</title>
      <composer>Fernandez</composer>
    </track>
    <track no="10">
      <title>Same Difference</title>
      <composer>Beckenstein</composer>
      <composer>Fernandez</composer>
    </track>
  </cd>
  <score>
    <title>Morning Dance</title>
    <group>Spyro Gyra</group>
    <composer>Beckenstein</composer>
    <guitar>C dim 7 ...</guitar>
  </score>
</music>
```

In the following we call this XML value *music*.

2.1 Querying XML: The Traditional Way

Two typical, simple queries on databases like the one shown above are to select all sub-elements of a certain kind or to get any sub-element that contains a specific piece of data. In this example we might be interested to obtain information about

1. All available scores
2. All material for one's favorite group

In the following we consider how such queries can be expressed in two textual XML query languages: Lorel and XML-QL. The Lorel query language [9] has evolved from a semi-structured database query language into an XML query language. Its syntax follows the tradition of SQL, but is has, of course, additional elements to account for nested objects.

The two tasks from above can be solved with Lorel quite easily. The first query reads as:

```
select music.score
```

The result is just the **score**-element. The second query uses a wildcard `%` for matching arbitrary tags of music sub-elements, and the restriction concerning the favorite group is expressed with a `where` clause.

```
select music.%
where music.%.group = "Spyro Gyra"
```

This query returns the **cd**- and the **score**-element. In both cases the result is wrapped in a root element <answer>. (To change the root-element to <scores> or <favorite> one has to use an as clause or an additional update.)

Both queries are very simple – at least for computer scientists or database experts, and for somebody who already knows SQL or something similar, Lorel is not difficult to learn and is certainly a good choice. (The same applies, in principle, to many other textual query languages that have been proposed.) However, end users typically do not have this technical background, and requiring them to learn a query language will be in most cases prohibitive. Hence, there must be a different way to provide end users with XML query capabilities.

Let us consider XML-QL [5] as another query language proposal to illustrate a further point. Although XML-QL is a textual query language, too, it differs from Lorel and other XML query languages by the approach to formulate queries by XML patterns, that is, a user can pose a query by describing *what* the result should be rather than how to obtain it. For example, the two queries about the music database are written in XML-QL as:

```
CONSTRUCT <scores> {
  WHERE
    <music>
      <score></score> ELEMENT_AS $s
    </music>
  CONSTRUCT $s
} </scores>
```

and

```
CONSTRUCT <favorite> {
  WHERE
    <music>
      <$a>
          <group>Spyro Gyra</group>
      </> ELEMENT_AS $x
    </music>
  CONSTRUCT $x
} </favorite>
```

These queries are definitely more verbose than the two Lorel queries, yet the nice point about this style is that conditions on the structure and contents of elements can be directly expressed by the patterns and need not be cast into path expressions or additional conditions. This makes it easy for somebody who knows XML to express queries on XML data. However, end users do not generally know about XML syntax, and they do not want to have to care about it either, in particular, because XML syntax with its angle brackets, slashes, and nested tags is not a very human-friendly notation.

Thus, as with the Lorel approach, the requirements on the technical background to use the query language present a usage barrier that is probably too high for most end users.

2.2 Design Goals for an End User Language

Driven by the above examples we have derived a couple of goals that serve as guidelines in the design of an XML end user query system.

1. Do not define a(nother) textual query language.
2. Avoid XML syntax.
3. Use a simple and intuitive visualization of XML.
4. Employ pattern matching.
5. Facilitate restructuring.
6. Keep the system as simple and intuitive as possible.

The first two items (that are somehow summarized in the third one) lead to using a visual language for XML. The language developed is based on a simple visualization of XML data, called the *document metaphor*, which is described in the next Section. Based on this we will introduce the concepts of *document patterns* in Section 4 and *document rules* in Section 5, and we will demonstrate how to formulate queries on XML data.

3 A Simple Visualization of XML

Recall the sample XML value *music* from Section 2.1. The grouping and structuring of information parts by nesting and bracketing tags is clearly unambiguous and serves as a precise specification of the structured data. In particular, it is well suited for being processed by computers, and since it is a text-based format, it is also well suited for being exchanged between different computer systems. However, it is not at all easily comprehended by humans. This is indicated by the fact that whenever XML data is used in communications, new lines and indentation is employed to visually emphasize the structure in the data. In fact, an end user will hardly ever see XML data in its raw form, but instead will look at a page properly rendered by a browser. Therefore, it is frustrating for users having to learn XML as a particular representation of data to pose queries.

Hence, we seek a visualization of XML data that is visually attractive, yet simple and intuitive, and that still reflects the structure of XML data. This means in particular that the visualization is in one-to-one correspondence to XML.

Our design is strongly influenced by the kind of documents people are used to deal with: faxes, product descriptions, all kinds of forms, and so on. These documents typically contain more or less portions of free text together with categorized informations (fields). Fields consist of a *header*, that is, a short textual description, and a *value*, which is given by text or another structured part, for example, a collection of further fields.

Now we essentially propose to represent elements (that is, fields) by boxes with the tag printed in bold face as a header above the box and the contents visualized inside the box. As an abbreviation for elements that contain only text we use “**tag**: *text*”. The same is used for attributes except that attribute names are not set in bold face. This representation mimics the layout of form-based documents, which is well-known to most people. This is especially supported by the abbreviating notation for unstructured elements.

For instance, the XML value *music* from Section 2.1 is visualized as shown in Figure 1.

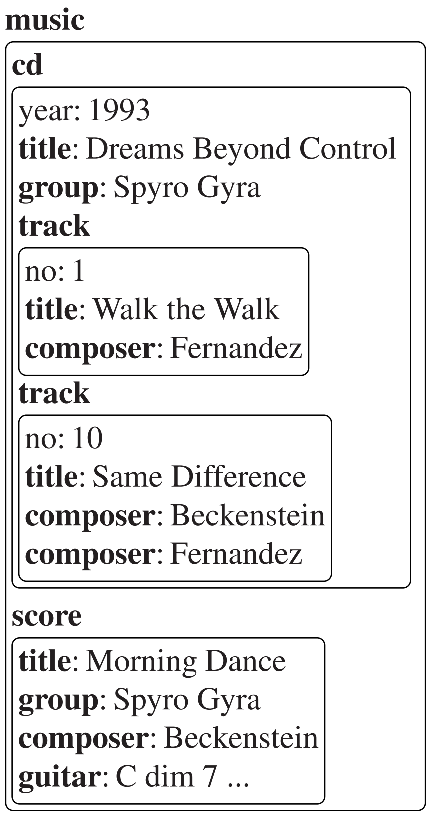

Figure 1: Document Visualization of Music Resources

Of course, this is not as nice as a tailor-made visualization, but it is definitely easier to understand (at least for end users) than the XML representation. Moreover, the very same visualization can be directly employed as a visual query language. This is what makes it interesting for end users: once a user knows how documents look like, she can use sample documents as patterns to extract information from XML data resources. There is no need to learn a query language or additional visual annotations to express queries. Moreover, by just using a second pattern (a result pattern) document rules can be formed that offer more control over the result and that can even express restructuring transformations of data. Document patterns and document rules make up the visual XML query and transformation language *Xing* [7] (which is pronounced "crossing" and which is an acronym for **X**ml **in** **G**raphics).

4 Document Patterns

Pattern matching provides a light-weight approach to data processing: it frees the user, to a large degree, from the need to use a language to express a search. Instead, the desired data is described by samples, or *patterns*, that specify structural and contents constraints. Therefore, pattern matching is suited very well as a paradigm for end user query languages.

Usually, a pattern consists of constants and variables; the constants specify objects that must appear in the data to qualify as a search result, that is, to match the pattern, and the variables are used to bind other data that is needed for further processing (and maybe for further constraints). Unfortunately, the concept of variables re-introduces the flavor of formal language, which is discouraging for most end users. This was striking in database query interfaces: systems that followed the Query-By-Example philosophy never became very popular, at least compared to form-based query facilities offered by many database vendors. One reason might be that QBE required the use of variables for certain tasks whereas the so-called "Query-By-Forms" interfaces offered the possibility of using data entry forms to pose simple queries just by entering sample data (without being explicitly exposed to a schema) – no new concepts had to be learned.

In the same way a document can be instantly viewed as a document pattern and can be directly used as a query. Though there are some extensions that make document patterns little more complex than documents, these are not required for simple queries, and even if needed, the extended notation can often be inserted automatically by the user interface, and the user does not necessarily need to learn it.

Let us now consider how the queries from Section 2.1 can be expressed in Xing. First, finding all scores can be achieved by the following simple pattern:

Figure 2: Find all Scores

This matches in the root element **music** all **score**-sub-elements.

Finding the resources of one's favorite group is a bit more complex. First, we want to match arbitrary sub-elements, that is, sub-elements having an arbitrary tag. For this we can employ regular expressions as tag names, in this case a simple wildcard matching any tag name. However, of all these sub-elements we want to match only those that have a sub-element **group** which itself has as content the text "Spyro Gyra".

Figure 3: Find Resources of Favorite Group

Using regular expressions is only one way to generalize (or weaken) queries to yield more results. Other possibilities are *or-patterns* and *deep queries*. An or-pattern combines two patterns and matches any element that matches either of the two patterns. An example is to refine the previous query to get only scores for guitars. This means that we cannot simply use a wildcard for matching all sub-elements of the **music**-value because we have different sub-criteria for different sub-elements. The notation for or-patterns is similar to the notation used by regular expressions, that is, different patterns can be joined by a bar. Hence we get:

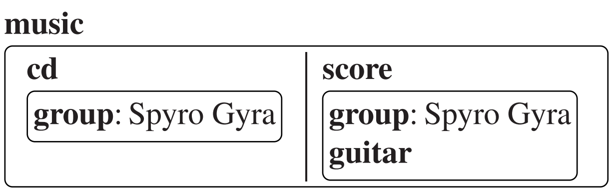

Figure 4: CDs and Guitar Scores of Favorite Group

More convincing examples for the use of or-patterns can be found in [7].

A deep query is given by any pattern that is prefixed by an ellipsis meaning to look for that pattern at any nesting depth within the searched document. An example is to find all pieces of a particular composer, see Figure 5.

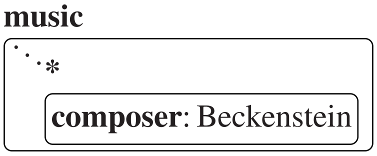

Figure 5: All Pieces of one Composer

Without the ellipsis, only the **score**-element would have been retrieved.

5 Document Rules

Document rules can be used to reformat or to restructure query results. Although the latter task might not be required by most end users, reformatting seems to be needed from time to time. A document rule consists of two document patterns that are joined by a double arrow: $P \Rightarrow Q$. P is called the *argument pattern* and specifies structural and content constraints. Argument patterns are responsible for the selection of the wanted data. Q is called the *result pattern* and specifies how the found elements are to be presented. This means, apart from doing restructuring, result patterns mainly perform projections on sub-elements.

For example, in the result of query shown in Figure 3, the **cd**- and **score**-element both contain only the **group**-element. The reason for this is that a document pattern alone is just a shorthand for a document rule in which the result and the argument patterns are the same, that is, P abbreviates $P \Rightarrow P$. In this case this leads to projecting just onto the group information. Therefore, to get more information about found elements a fully-fledged pattern rule with an appropriate result pattern must be used. A simple solution is to use just the element name without any sub-elements in the result pattern. But here we face another problem: since we have used a wildcard in the argument pattern, we do not know which element name to use in the result pattern. We could use the wildcard itself, but this can lead to ambiguities if multiple wildcards are used in an argument pattern. Therefore, we require to assign an additional name to any found element, called an *alias*, which can be referred to by the result pattern.

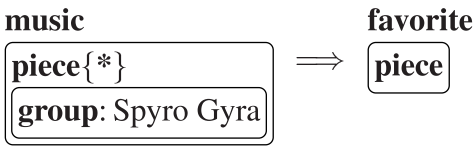

Figure 6: Resources of Favorite Group

This query will return the complete XML data source. Of course, if we want to see, for example, just the title and the group, we can add the corresponding sub-elements to the result pattern.

To illustrate data restructuring we consider the task of building a list of titles and composers. In the argument pattern we again use a deep pattern to get titles of scores as well as track titles, and in the result pattern we simply omit any grouping elements around the title- and composer-elements, see Figure 7. This has the effect of flattening XML-trees which is done by computing a cartesian product of the lifted elements.

Figure 7: List of Titles and Composers

The necessary restructuring operations can be always identified by comparing the result with the argument pattern: omitting elements means to apply flattening/cartesian product operations, inserting new elements results in group-by operations. For example, if wanted to have a list of composers together with all their titles, we can insert a **titles**-element around the **title**-element in the result pattern.

6 Exploiting DTDs for the User Interface

A Document Type Definition restricts the use of attributes and the possible nesting of elements in XML data; it defines a range of possible instantiations for XML values. We have deliberately made the query mechanism independent from DTDs to keep it simple and generally applicable. However, if present, DTDs provide useful information about the data source that should be taken advantage of. One highly useful application is in the user interface. A user must somehow construct document patterns to build up XML queries. Typically, this happens by an editor that allows to create boxes and to assign and change box tags.

Now having a DTD available, creating named boxes can be made very easy: a pop-up-menu can offer all possible attributes/sub-elements of the box clicked in. From this menu one or more tags can be selected, and a nested box together with an appropriate label can be automatically inserted. When two or more elements are selected, a regular expression would be constructed. Moreover, as mentioned above, the wildcard can be inserted automatically if all tags are selected in the menu.

Repeatedly creating sub-boxes in one and the same box B results in adding sub-boxes to B, that is, adding sub-elements to the pattern represented by B. In contrast, or-patterns can be built by marking a sub-box before invoking the menu. Altogether we can obtain a relatively simple point-and-click user interface to create a large variety of queries in a simple and easy-to-learn way.

Beyond the user interface aspect, DTDs also provide automatic and incremental static type checking of patterns and rules. Moreover, DTDs can also be utilized to optimize queries and transformations.

7 Related Work

There is large selection of XML query languages. We have already discussed XML-QL [5] and Lorel [9], but there are many more, for instance, YatL [3], and XQL [12] to give just two further important examples. These all are dedicated XML query languages. Also related are languages for querying the World Wide Web, for a survey see [8], and some proposals from the area of document processing, for example, [11] or [10]. All the above mentioned approaches are text-based which make them difficult to use for end users.

However, there are also a couple of proposals for visual interfaces for querying XML: in XML-GL [2] queries are denoted by drawing trees that reflect the structure of the queried XML data. XML-GL is based on the fact that XML expressions are in essence multi-way trees. However, trees represent rather an abstract visual syntax of XML documents, which is well suited for formal language manipulations [6] but not necessarily for end user query languages. From this point of view, XML-GL is more like general-purpose visual programming languages.

In contrast, our proposal to represent XML documents in a form-like way by nested rectangles with attached labels achieves a higher degree of usability, in particular, in the sense of *concreteness* [1] and *directness* [13].

A form-based query interface is also provided by EquiX [4]. However, the form metaphor is only used on the outermost level, and nesting is expressed by simple indentation. The forms are generated semi-automatically, driven by a DTD. This means a severe restriction since only data sources providing a DTD can be queried. Moreover, the expressiveness of EquiX is quite limited: it is not even possible to reformat the query results, and it is not possible either to perform data restructuring or deep queries.

8 Conclusions

We have designed a form-based interface to denote XML queries and transformations. Based on a visual document metaphor and the notion of document patterns and rules, it provides a direct and simple way to locate and select information in XML data resources.

We believe that the described interface is usable by a broad audience because (i) the underlying document metaphor reflects common knowledge about forms, (ii) the notion of document pattern is immediately accessible, and (iii) the interface is completely independent from a complex, textual formal query language.

References

[1] M. M. Burnett. Visual Programming. In Webster, J. G., editor, *Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1999.

[2] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *8th Int. World Wide Web Conference*, 1999.

[3] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD Conf. on Management of Data*, pages 177–188, 1998.

[4] S. Cohen, Y. Kanza, Y. Kogan, W. Nutt, Y. Sagiv, and A. Serebrenik. EquiX – Easy Querying in XML Databases. In *2nd ACM SIGMOD Int. Workshop on The Web and Databases*, pages 43–48, 1999.

[5] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *8th Int. World Wide Web Conference*, 1999.

[6] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

[7] M. Erwig. A Visual Language for XML. In *16th IEEE Symp. on Visual Languages*, pages 47–54, 2000.

[8] D. Florescu, A. Levy, and A. O. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[9] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *2nd ACM SIGMOD Int. Workshop on The Web and Databases*, pages 25–30, 1999.

[10] P. Kilpeläinen and H. Mannila. Retrieval from Hierarchical Texts by Partial Patterns. In *16th ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 214–222, 1993.

[11] M. Maruta. Data Model for Document Transformation and Assembly. In *4th Int. Workshop on Principles of Digital Document Processing*, LNCS 1481, pages 140–152, 1998.

[12] J. Robie, editor. *XQL (XML Query Language)*, 1999. `http://metalab.unc.edu/xql/xql-proposal.html`.

[13] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69, 1983.