Xing: A Visual XML Query Language

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

Abstract

We present a visual language for querying and transforming XML data. The language is based on a visual document metaphor and the notion of document patterns and rules. It combines a dynamic form-based interface for defining queries and transformation rules with powerful pattern matching capabilities and offers thus a highly expressive visual language. The design of the visual language is specifically targeted at end users.

Key Words: XML, End User, Query Language, Document Metaphor, Document Transformation

1 Introduction

The eXtensible Markup Language (XML) is a standardized notation for documents and other structured data [3]. XML allows the definition of specialized markup languages for any domain. XML is an emerging standard for data exchange, and it is very likely that the vast amount of tomorrow's data and web resources will be 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.

A query interface to XML documents must address several important issues. Two aspects that have a great influence on favoring a visual over a traditional textual notation are:

1. 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 [3], XPath [11], XSLT [10], and so on.

   An end-user query interface has to hide all these technically involved, although important, aspects from the user.

2. Search engines that are solely based on key words are not powerful enough to exploit the structure that the XML format contributes to data. Everybody who has used search engines knows how difficult it is to choose keywords that lead to reasonably sized and useful results.

   Dedicated query languages definitely can help in the case of XML to find keywords in particular contexts. However, query languages are not an option for most end users who even have no knowledge of what a language with keywords, variables, and so on, is. Therefore, an end-user query system must avoid exposing a formal language to the user. For example, experience with relational databases has shown that form-based query and update facilities rather than, say, SQL, are accepted and employed by end users.

The universal character of XML as a data wrapping tool and its envisioned omnipresence on the Internet cause a specific need for XML query systems targeted at end users. The impact of designing and implementing such interfaces is very high since end users make up such a huge group of XML clients. Therefore, the visual language should be easily accessible even to novice computer users.

In this paper we propose a visual query and restructuring language for XML, called *Xing* (which is pronounced “crossing” and which is an acronym for **X**ml **in G**raphics). A Xing program is, in general, given by a *rule*, and a rule consists of two *patterns*. A pattern is a kind of visual sketch of an XML document, and this visualization of XML data forms the basis of the design of Xing. The design of the XML visualization is driven by the observation that many of the documents people are used to deal with are typically organized into fields into which text or other structured data, for example, a collection of further fields are to be entered. Our visual representation mimics exactly these kinds of documents, which are well-known to most people.

A particular support for end users is also obtained by the general design of Xing because they can immediately start by expressing simple queries (which are just given by single patterns): one only has to draw a sketch of the documents/data one is looking for. Moreover, drawing such a sketch can be well supported by the user interface so that queries can eventually be constructed just by selections made from some pop-up menus. In general, end users can express queries with just the knowledge of what (visually) constitutes a document. In particular, no concept of things like keyword, variable, etc., is needed on the part of the user. The reason for this is that fields are implicitly interpreted as variables, text constants work as selections, and the act of mentioning or omitting fields is additionally interpreted as projection on data.

Of course, to pose more advanced queries, in particular, to perform complex data restructuring, some more abstract concepts of Xing, such as wildcards or rules, have to be realized, but as we will see, there are only few simple syntactic rules that have to be learned. It might seem that the notions of “end users” and “advanced queries” contradict each other. For example, one might argue that data restructuring might not be needed at all by end users, but this strongly depends on the data source and its internal structure. (It also depends on what exactly is considered to be an end user.) For example, highly structured data sources are more likely to require restructuring than flat lists of information. In general, it seems that the longer a user works with a system, the more functionality he or she explores and demands. Hence, we believe that data restructuring should be part even of an end-user query system to not disappoint users that become comfortable with the system and are expecting more functionality.

In the next section we give a short introduction to XML, in particular, we consider how to query XML documents. Section 3 presents the visual language in three steps: first an XML visualization called the *document metaphor* is introduced in Section 3.1. After that document patterns and rules are presented in Sections 3.2 and 3.3. Section 4 gives a semantics of document patterns and rules mostly by informal explanations and examples. The formal semantics definitions can be found in the Appendix. In Section 5 we consider several extensions to the basic notation, and in Section 6 we explain how DTDs (document type definitions) can be exploited for the user interface. Related work is discussed in Section 7, and conclusions are given in Section 8.

2 XML

XML is a text-based format to represent structured documents and data. Some of XML’s most important features are:

- *Simplicity*. Structure information is obtained by named opening and closing tags. As a text format, XML can be easily read, produced, and transmitted.
- *Generality*. From its general structure any desirable data format can be instantiated.
- *Standardization.* The XML standard is developed and controlled by the World Wide Web Consortium. Many large and important companies are supporting XML.

Altogether, these characteristics make XML a first choice data format for data exchange and representing data on the Internet. In the following we will first give a short introduction to XML in Section 2.1. We then demonstrate in Section 2.2 how XML data can be queried with existing textual query languages. From this we derive design goals for an end-user query system in Section 2.3.

2.1 XML Documents

The structure of an XML document is obtained through the use of tags, which mark the beginning and ending of so-called *elements*: an element starts with a tag, for example, `<book>`, is followed by an arbitrary sequence of other elements (for example, title and author elements) and text fragments, and ends with a corresponding tag, in this case: `</book>`. Elements might also contain attributes, which are then included as *name*`="`*text*`"`-pairs in the start tag of the element, as in, for example, `<book year="1988">`. In contrast to sub-elements, the order in which attributes appear in an element does not matter.

If all begin and end tags are properly nested in an XML document, the document is said to be *well-formed*. Apparently, well-formed XML documents organize their contents in a hierarchical way and thus essentially describe trees. An XML document is given by just a single XML element which is also called the *root element*.

Let us consider a small example. Below we show an XML document containing some bibliographic data.

```
<bib>
  <book year="1988">
    <title>Concrete Mathematics</title>
    <author>Graham</author>
    <author>Knuth</author>
    <author>Patashnik</author>
  </book>
  <article year="1998">
    <title>Linear Probing and Graphs</title>
    <author>Knuth</author>
    <journal>Algorithmica</journal>
  </article>
</bib>
```

We call this XML element *bib*. The root element contains two further elements, a `<book>` and an `<article>` element, which both have a year attribute and contain further elements. Since their contents is structured by sub-elements, these two elements (as well) as *bib* are called *structured elements*. In contrast, for example, the `<title>` and `<author>` elements are called *non-structured elements* since they contain only text and no sub-elements.

The structure of an XML document can be constrained by a so-called "Document Type Definition" (DTD) which specifies all possible elements, their attributes, and their allowed nestings. A DTD represents a kind of type information for XML values. An XML document conforming to a DTD is said to be *valid*.

We deliberately ignore DTDs at the user level for two reasons: first, many information that is currently available on the Web will be transformed into XML without conforming to or even having a particular DTD; this will be particularly true for large amounts of unstructured data. It should be possible to query all these information sources with the proposed language, however, relying on the presence of DTDs could be in many cases prohibitive. Second, we do not want to require users of the interface to have knowledge of the concept of a "schema". Nevertheless, there are ways to nicely exploit DTDs when they are given. In particular, the user interface can greatly benefit from DTDs. We shall discuss this issue later in Section 6.

2.2 XML Query Languages

Recall the XML element *bib*. Two simple, typical queries on such a document are to select all sub-elements of a certain kind or to get any sub-element that contains a specific piece of data. A more advanced, but also quite common task is to regroup elements according to specific sub-elements they have in common. In this example we might be interested to find:

Query 1: All books

Query 2: All publications of a particular author

Query 3: A list of publications grouped by authors

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 [26] has evolved from a semi-structured database query language into an XML query language. Here the term "semi-structured data" refers to data that is not necessarily strongly typed and that sometimes does not have an explicit schema at all [4, 23]. Note that "Lorel" is the name of the query language and "Lore" is the name of the underlying database system. Lorel's syntax follows the tradition of SQL, but is has, of course, additional elements to account for nested objects. In particular, Lorel contains path expressions that can be used to navigate through hierarchical objects. Query 1 is expressed in Lorel as follows:

```
select bib.book
```

The result is just the `<book>` element:

```
<book year='1988'>
  <title>Concrete Mathematics</title>
  <author>Graham</author>
  <author>Knuth</author>
  <author>Patashnik</author>
</book>
```

If we wanted to store all the found books for further processing in a new element, say `<books>`, we had to use the update mechanism of the current Lore system, which is is a bit clumsy for this purpose:

```
books := oem(dummy:"");
update books.dummy := (select bib.book);
```

The function `oem` creates a new root element, and `dummy` is used to receive the contents of that element. This mechanism has to be used because it is not possible in the current Lore system to rename root elements of query results.

The second query uses a wildcard `%` for matching arbitrary tags of bib sub-elements, and the restriction concerning the author is expressed with a `where` clause.

```
select bib.%
where bib.%.author = "Knuth"
```

This query returns both, the `<book>` and the `<article>` element. Again, to wrap the results in an additional element we had to use the update trick shown above.

Whereas the first two queries were simple selections, the third task requires a restructuring of the data because in the data source the bibliographic data is grouped by publications whereas we require the data grouped by author. In the current Lorel system (Version 5.0) it is not possible to express the third query. The envisioned future query language allows more easily to construct new XML elements and allows to express cartesian product, join, and regrouping by nested queries. The query for the third task would be as follows:

```
select xml(byAuthor:
  (select xml(pub:{author: a,
                   (select bib.%.title
                    from bib.% p
                    where p.author = a)})
   from bib.%.author a))
```

This query expresses a join on `<author>` elements and puts the titles for each author (found by the innermost `select` expression) directly after the corresponding `<author>` element. The result in this case is:

```
<byAuthor>
  <pub>
    <author>Graham</author>
    <title>Concrete Mathematics</title>
  </pub>
  <pub>
    <author>Knuth</author>
    <title>Concrete Mathematics</title>
    <title>Linear Probing and Graphs</title>
  </pub>
  <pub>
    <author>Patashnik</author>
    <title>Concrete Mathematics</title>
  </pub>
</byAuthor>
```

Let us consider XML-QL [16] 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 look like rather than how to obtain it. For example, the first query about the *bib* database is written in XML-QL as:

```
CONSTRUCT
<books> {
  WHERE
    <bib>
      <book></book> ELEMENT_AS $b
    </bib>
  CONSTRUCT
    $b }
</books>
```

The outermost `CONSTRUCT` builds a new root element that contains the result of the enclosed `WHERE`/`CONSTRUCT` part. The `WHERE` clause performs the selection by matching against the `<bib>` element and within that matching all `<book>` elements. Each found element is then bound to the variable `$b` with the `ELEMENT_AS` clause. This variable is then referenced in the following `CONSTRUCT` clause that actually builds the result and causes all found books to appear in the newly created `<books>` element.

The second query uses a tag variable `$p` to match any element within the `<bib>` element. Note that the closing tag of any tag variable is just `</>`. Then with the nested `<author>` element the condition is expressed that only elements that have Knuth as an author should be matched. As in the previous query all qualifying elements are bound to a variable that is used in the `CONSTRUCT` part to create the result.

```
CONSTRUCT
<knuth> {
  WHERE
    <bib>
      <$p>
        <author>Knuth</author>
      </> ELEMENT_AS $k
    </bib>
  CONSTRUCT
    $k }
</knuth>
```

The query for the third task matches any bibliographic entries as in the above query, but without the restriction that the author is Knuth. In the `CONSTRUCT` clause a new `<pub>` element is then produced for each `<author>` element, and a nested query finds all titles for that author. This join is expressed by using the very same variable `$a` inside the inner `WHERE` clause that was bound by the outer `WHERE` clause.

```
CONSTRUCT
<byAuthor> {
  WHERE
    <bib>
      <$p>
        <title>$t</title>
        <author>$a</author>
      </>
    </bib>
  CONSTRUCT
    <pub>
      <author>$a</author> {
      WHERE
        <bib>
          <$p>
            <title>$t</title>
            <author>$a</author>
          </>
        </bib>
      CONSTRUCT
        <title>$t</title> }
    </pub> }
</byAuthor>
```

The XML-QL queries are definitely more verbose than the 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. Moreover, it is somewhat easier to create new XML elements.

Altogether, the pattern matching approach makes it relatively easy for somebody who knows XML to express queries on XML data. However, to use XML-QL one needs to know about XML syntax, which is not a very user-friendly notation with all its angle brackets, slashes, and nested tags.

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

2.3 Design Goals

Reviewing the above query examples we can now derive some design goals for an XML end-user query system. First of all, we should not define a(nother) textual query language.

Second, concrete XML syntax should be avoided, too. A few levels of nesting can easily lead to confusion about the opening and closing tags. This is indicated by the fact that whenever XML data is used in communications among humans, 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, we instead employ a visualization of XML. Ideally, the same visualization could be used for presenting XML data and for the representation of queries.

Third, since pattern matching is a powerful concept that greatly supports the examination of structured data, pattern matching should be employed in the query system. This allows the specification of queries mainly without a dedicated query syntax by just giving examples. Brought into a form-based context, the pattern matching style of querying can also be well supported by simple point-and-click user interfaces.

Finally, as a general design rule, the system should be as simple as possible. This seems to be an obvious requirement, but it can be supported by a particular design attitude, namely to make simple queries very easy and impose more work and concepts on advanced queries.

Altogether, these design considerations lead to using a rule-based 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.

3 Visual XML Querying

The language Xing we have developed is based on a simple visualization of XML data, called the *document metaphor*, which is described in Section 3.1. Based on this we will introduce the concepts of *document patterns* in Section 3.2 and *document rules* in Section 3.3, and we will demonstrate how to formulate queries on XML data. Although we will describe the treatment of attributes, our emphasis is on how to deal with elements. This helps to keep the presentation more focused.

3.1 A Visual Document Metaphor

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 has to be 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, purchase orders, bank statements, all kinds of forms, and so on. These documents typically contain more or less portions of free text together with fields into which data can be entered. 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*". This notational convention greatly simplifies the visual appearance of documents: a whole level of nested boxes can be saved. This lowers the visual complexity and eases the understanding of pictures. The same notation is also used for attributes except that attribute names are not set in bold face. This similarity in notation was chosen deliberately because of the already mentioned similarity of attributes and unstructured elements. Altogether, 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 *bib* from Section 2.1 is visualized as shown in Figure 1.

The order of the sub-elements is given by their relative vertical position (recall that the order of attributes does not matter).

Of course, this visualization is not as nice as a tailor-made layout, but it seems to be easier to comprehend than the

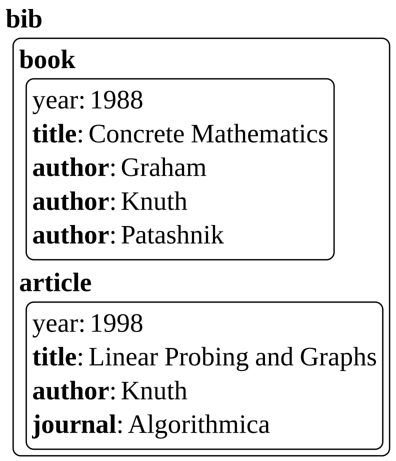

Figure 1: Bibliography as a Xing expression

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 results and that can even express restructuring transformations of data.

3.2 Document Patterns

Pattern matching provides a lightweight 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 seems to be 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 (as well as always exposing the complete relation schema) 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 explicitly exposing 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. Let us now consider how the queries from Section 2.2 can be expressed in Xing. First, finding all books can be achieved by the pattern shown in Figure 2:

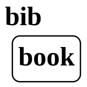

Figure 2: Find all books

This pattern just matches all **book** elements that are direct sub-elements of the root element **bib**. The result of this query is shown in Figure 3.

Note that the root element’s tag of the result is **bib**. To get the books wrapped into a **books** element we have to use document rules, which will be introduced in the next section.

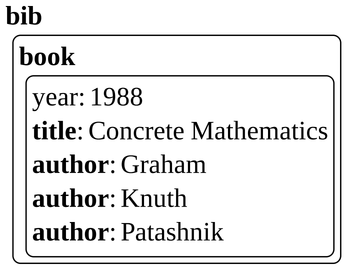

Figure 3: Query result: all books

Retrieving all publications of a particular author 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 **author** which itself has as content the text “Knuth”; see Figure 4.

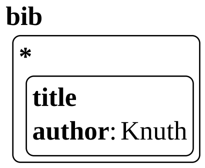

Figure 4: Find all publications of Knuth

Using a wildcard, which is just a special case of regular expression, is only one way to generalize (or weaken) queries to yield more results. We have chosen “*” as in XPath [11]. Other possibilities are *or-patterns* and *deep queries*. We will consider these in Section 5.

It might seem that requiring end users to employ regular expressions in queries is too challenging. However, there is some empirical evidence that a combined textual/visual representation of regular expressions can be successfully utilized by end users [2, 30]. We plan to investigate improvements of the query notation in that direction.

The third query cannot be expressed with just a document pattern, because it asks for restructuring of data. Expressing restructuring requires document rules, which we will consider next.

3.3 Document Rules

Document rules can be used to restructure query results. 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 desired 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.

Consider, for example, the result for the query from Figure 4, which is shown in Figure 5.

The result might be somewhat surprising since the reader might have expected all authors of the book to appear in the result and not just Knuth. Likewise, one might ask why the journal is not shown in the **article** 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 title and author information. Therefore, to get more information about found elements, a document 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. An alias is written as a tag followed

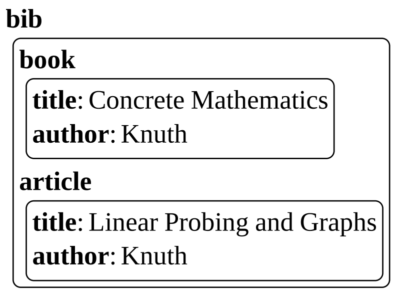

Figure 5: Query result: all publications by Knuth

by the regular expression in curly brackets. In the example, we can also use the opportunity to name the root element differently.

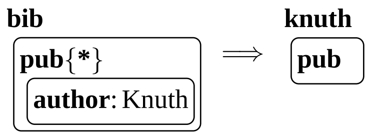

Figure 6: More details about Knuth's publications

This query will actually return just the document *bib* with the root element changed to **knuth**. Further refinements of this query will be discussed in Section 5.

The fact that a user is required to employ an additional result pattern to find, for example, the full information about Knuth's publications, is unfortunate, although this design keeps the query language simple and leads to a concise and consistent semantics. One possible improvement is to use a different color (or another marking mechanism) for tags that should be used only for selection and not for projection. Due to lack of color, we use a bar "|" preceding a sub-pattern to indicate that it should just be used for selection. With this extension the query from Figure 6 can be expressed by using just a single a pattern; see Figure 7.

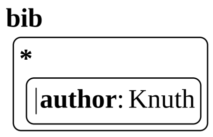

Figure 7: Knuth's publications found by using an extended pattern

The result differs from that of the previous query only in the name of the root element that is still **bib**. With this extension we can express subtle variations of queries: for instance, by putting the bar in front of "Knuth" we find all authors of all publications having Knuth as a co-author. This extension could even be used to emulate existential patterns described in Section 5. Consider, for example, the query asking for titles and authors of all publications of which Knuth is a co-author. We can express this by the pattern shown in Figure 8.

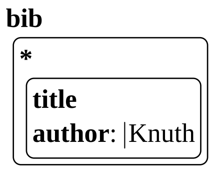

Figure 8: Titles and authors of Knuth's publications

The same query expressed with existential patterns is shown in Figure 17. The meaning of an extended pattern *p* is the same as the rule $p \Rightarrow q$ where q is obtained from p by removing all sub-patterns marked with a bar.

Returning to the issue of document rules, another example is to modify the book query from Figure 2 to yield only book titles. This can be done most easily by just adding a sub-element **title** to the **book** element. The result of such a query is shown in Figure 9.

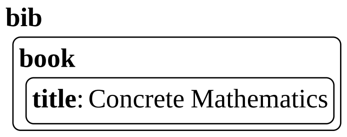

Figure 9: All book titles

Here, another question arises: should the year information be shown even though it was not explicitly requested by the result pattern of the query? One reason for including attributes by default could be that attributes are more closely tied to elements than sub-elements. On the other hand, in the interest of the simplicity of the query system attributes should not be treated in a special way. This means attributes should not be returned in query results if there is an explicit projection in a result pattern.

In the previous query, the **book** element is used to group only one title at a time and might be considered verbose. To remove it we essentially need a document rule to express the desired result. We can obtain a flat list of book titles by adding the **title** element to the argument pattern and by removing the **book** element that encloses the **title** element in the result pattern. The query is shown in Figure 10.

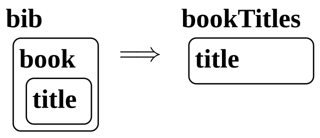

Figure 10: Flat list of book titles

This query performs a restructuring operation: the argument pattern selects all titles nested within **book** elements, and the result pattern – by omitting the **book** element – describes a flattening of the found elements, which means here to concatenate all **title** elements from all books (and link these titles as direct children to the root element **bookTitles**). In general, “flattening” means to reduce the nesting depth of the XML value by one level.

On the other hand, “grouping” can be considered the inverse operation of flattening: a list of elements is transformed into a list of lists on the basis of common values for one (or more) sub-elements. As an example, assume we are interested in the list of publications grouped by authors. We can use the document rule shown in Figure 11 to accomplish this task.

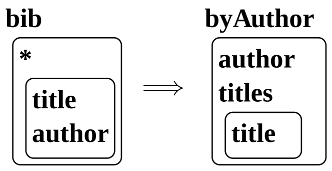

Figure 11: Authors and their publications

The argument pattern specifies the selection criterion, in this case to collect from all sub-elements in *bib* all author and title information. The result pattern then describes how the found information is to be structured. Here the intention is to list all authors, each followed by the titles of his publications. These titles are wrapped in a new **titles** element. In Figure 11 it is striking that the nesting structure of the result pattern is considerably different from the argument pattern. In particular, we can observe the following differences compared to the argument pattern:

(a) The nesting depth of the **author** tag has been reduced, and the * tag has disappeared.

(b) The *relative* nesting of the **title** tag with respect to the **author** tag has increased, and a new **titles** tag has appeared on the same level as **author**.

In fact, these pattern differences are exploited to determine the required restructuring operations. For instance, the first observation leads to the *flattening* of the elements matching the wildcard "*". The second observation is interpreted to group titles by authors and leads to the application of a corresponding *grouping* operation. The details of the query semantics will be explained in the next section.

4 Semantics

To define the meaning of a Xing query we need an abstract representation of documents as well as document patterns and rules. We use multi-way trees for this purpose. (Since documents are nothing but XML values, we could also use XML itself as a uniform representation. However, the linear textual representation is not suited well for formal manipulations.) We also define a couple of auxiliary operations for transforming XML values. This is done in Section 4.1. In Section 4.2 we use these operations to define the semantics of patterns and rules and demonstrate the semantics with an example. The formal definitions can be found in the Appendix.

4.1 Basic Definitions

XML Values

We basically use the notation that was introduced by XDuce [27, 29] (a similar notation is also used in Yatl [12]) and has been also adopted the W3C XML Query Working Group [24]: an element with tag t and sub-elements $e_1, \ldots, e_n$ is written as

$$t\,[\,e_1, \ldots, e_n\,]$$

where a tag, as well as text, is a string over some alphabet L. Note that $t\,[\,e_1, \ldots, e_n\,]$ denotes a *tree* (and not a list) and that each e_i is, in general, itself a tree. As an example, consider the XML value *bib* (shown in Figure 1), which is written in the above tree notation as:

$$\begin{array}{l}\textbf{bib}\,[\,\textbf{book}\,[\,\text{year}\,[\,1988\,],\\ \qquad\qquad \textbf{title}\,[\,\text{Concrete Mathematics}\,],\\ \qquad\qquad \textbf{author}\,[\,\text{Graham}\,],\\ \qquad\qquad \textbf{author}\,[\,\text{Knuth}\,],\\ \qquad\qquad \textbf{author}\,[\,\text{Patashnik}\,]\,],\\ \qquad \textbf{article}\,[\,\text{year}\,[\,1998\,],\\ \qquad\qquad \textbf{title}\,[\,\text{Linear Probing and Graphs}\,],\\ \qquad\qquad \textbf{author}\,[\,\text{Knuth}\,],\\ \qquad\qquad \textbf{journal}\,[\,\text{Algorithmica}\,]\,]\,]\end{array}$$

The described tree notation is more powerful and less trivial than it might look: the comma acts as an overloaded concatenation operator on lists (of elements), that is, the comma concatenates two lists, adds an element to either end of a list or takes two single elements and combines them into a two-element list. The use for this will become clear later. Since the comma concatenates lists, we need an additional notation to denote a list of lists. We use the semicolon for this purpose. Hence, whereas xs, ys denotes the (flat) list of all elements from xs and ys, $xs; ys$ denotes a two-element list of lists. The empty list is denoted by $()$. We sometimes abbreviate an empty element $t\,[\,()\,]$ (that is, a tree node that has no children) by just writing t.

Patterns

Since patterns are basically documents, we can use the same abstract tree representation as for XML values. For instance, the pattern of the query from Figure 2 is given by:

$$\textbf{bib}\,[\,\textbf{book}\,]$$

Wildcards can be expressed by using corresponding tag names. (Other extensions like deep patterns could be introduced as syntactic extensions as needed.) Finally, a rule is simply given by a pair of patterns. For example, the abstract representation of the query shown in Figure 11 is given by the pair (p,r) where

$$\begin{aligned} p &= \mathbf{bib}[*[\mathbf{title},\mathbf{author}]] \\ r &= \mathbf{byAuthor}[\mathbf{author},\mathbf{titles}[\mathbf{title}]] \end{aligned}$$

The semantics of document patterns is expressed on the basis of selection and projection operations on XML values. The additional restructuring power of document rules requires a flattening and a grouping operation on XML values for the definition of the semantics.

Therefore, we first define the basic operations and explain the semantics of document patterns and rules in terms of these operations afterwards.

Marked XML Values

To decompose the semantics description into manageable pieces we introduce the notion of *marked XML values*. A *marked XML value* is written with a small dot preceding it, and marked and unmarked XML values can contain both marked and unmarked sub-elements, for example, $\cdot t[u[\cdot v[a]],w[b]]$. With marked XML values we can perform selection and projection in two independent steps. For example, we can obtain the titles of all books written by Knuth by the query shown in Figure 12.

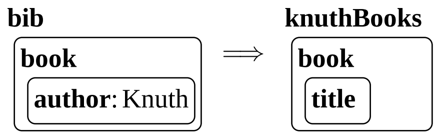

Figure 12: Titles of Knuth's books

Now this query is processed (in the semantics) in two steps: first, the left-hand side pattern marks all books that have an **author** element that contains the text "Knuth", that is, the intermediate result is the marked XML value:

$$\begin{aligned} &\mathbf{bib}[\cdot\mathbf{book}[\text{year}[1988],\\ &\qquad\mathbf{title}[\text{Concrete Mathematics}],\\ &\qquad\mathbf{author}[\text{Graham}],\\ &\qquad\cdot\mathbf{author}[\cdot\text{Knuth}],\\ &\qquad\mathbf{author}[\text{Patashnik}]],\\ &\quad\mathbf{article}[\text{year}[1998],\\ &\qquad\mathbf{title}[\text{Linear Probing and Graphs}],\\ &\qquad\mathbf{author}[\text{Knuth}],\\ &\qquad\mathbf{journal}[\text{Algorithmica}]]] \end{aligned}$$

Second, a projection with the right-hand side pattern on this intermediate value is performed, which selects only the marked **book** elements and projects onto the **title** elements of these. Finally, all marks are removed. Hence, the final result of the query is just:

$$\mathbf{bib}[\mathbf{book}[\mathbf{title}[\text{Concrete Mathematics}]]]$$

The marks on the intermediate XML values are necessary to remember the selection made by a LHS pattern in the projection performed by a RHS pattern. On the other hand, two patterns and two "passes" over the XML value are necessary to allow for the selection criteria to be independent of the projection.

Matching and Selection

Matching a pattern against an XML value is a basic operation that is needed for extracting sub-values as well as for constructing results. A pattern $t[x_1,\ldots,x_n]$ matches an element $u[y_1,\ldots,y_m]$ if the tags t and u (string-)match and if there is at least one pattern x_i that matches a corresponding element y_i. If there are no sub-patterns (x_i), only a matching tag is required.

The selection operation takes an XML pattern and an XML value and returns the value with those parts marked that match the pattern. Here we find many different possibilities to define how the selection operation uses a pattern to filter a sequence of elements. It is out of the scope of this paper to investigate all the different options in detail. However, to motivate our design decision we briefly discuss the following aspect: suppose we match a list of patterns x, y against a sequence $z_1, \ldots, z_n$. Does this mean to mark:

1. all elements z_i that match x or y (this strategy would be a kind of independent double selection), or
2. the first subsequence of consecutive elements matching x followed by the first subsequence of consecutive elements matching y (this strategy does not ignore the order in the sequence pattern and is more like matching regular expressions)

For example, matching x, y against $a, x, x, a, x, y, x, b, y$ yields the following two different results under the two described semantics:

1. $a, \cdot x, \cdot x, a, \cdot x, \cdot y, \cdot x, b, \cdot y$
2. $a, \cdot x, \cdot x, a, x, \cdot y, x, b, y$

These two semantics somehow mark the two extremes on a landscape of possible interpretations, and there are many more variations in between that also make sense. For example, we can consider the following two variations for the second strategy. Allowing repetition of the whole pattern leads to:

3. $a, \cdot x, \cdot x, a, x, \cdot y, \cdot x, b, \cdot y$

or not requiring consecutive matching elements yields (without repetition):

4. $a, \cdot x, \cdot x, a, \cdot x, \cdot y, x, b, y$

Among the described possibilities there is no obvious “best” semantics. We have chosen yet another variation:

5. repeatedly mark the first subsequence of (not necessarily consecutive) elements matching x followed by the first subsequence of (not necessarily consecutive) consecutive elements matching y

In the shown example, this strategy yields the same result as the first option.

Hence, applying selection with a pattern $t[x_1, \ldots, x_n]$ to an element $e = u[y_1, \ldots, y_m]$ basically returns the element $\cdot e$ if it matches the pattern and e otherwise. More precisely, the sub-elements of e will undergo a selection, too, that is, selection will try to mark the elements $y_1, \ldots, y_m$ by trying to (repeatedly) match the sub-patterns $x_1, \ldots, x_n$.

Projection

The projection operation takes an XML pattern and a marked XML value and returns those parts of the value that are marked and match the pattern. All marks are dropped from the result (which is easily achieved by applying a corresponding unmark function to all parts of an element, see Appendix).

The definition of projection essentially tries to extract the pattern from the value to which it is applied. Note that although being seemingly very similar to selection, projection differs in two important ways:

- In addition to having a matching tag the value under consideration also has to be marked to qualify for the result.
- If a marked element contains unmarked sub-elements, these *are* included in the result regardless of their marks (as long as there are no sub-patterns explicitly asking for matching them).

We have already shown examples for selection and projection in the description of marked XML values.

Below we will see that the Xing semantics is defined essentially in two steps: first, elements matching the argument patterns are selected and marked as such, and second, these marked elements are projected using the result pattern. The similarity in the definitions for selection and projection makes it possible to use a single pattern as an abbreviation for a rule, that is, it is possible to use an argument pattern also in a reasonable way as a result pattern. On the other hand, it is the differences in the definitions that make rules a non-trivial extension to patterns and give control over restructuring.

Flattening

We illustrate the process of flattening by an example. Essentially, "flattening" means to eliminate one level of nesting and lift elements up one level, that is, an element like

$$t[u_1[e_{11},\ldots,e_{1k_1}],\ldots,u_n[e_{n1},\ldots,e_{nk_n}]]$$

is transformed by flattening into a new element

$$t[e_{11},\ldots,e_{1k_1},\ldots,e_{n1},\ldots,e_{nk_n}]$$

On a closer look, the definition of flattening raises two important questions: (i) which level should be eliminated and (ii) how should the remaining elements be integrated into the rest of the element? Concerning the first question, we define flattening to eliminate the level beneath the root element. Then we can still define operators that apply the flattening operator at any level in the tree. Regarding the second question, we compute for each element to be removed combinations of its sub-elements and then concatenate all these combinations for all removed elements. Now what does "combinations of sub-elements" mean? These combinations are comparable to a kind of cartesian product, and they are obtained by computing elements that contain exactly one sub-element for each of the original sub-element tags. Repeated elements for one tag lead to a generation of new elements, one element for each combination of values for each tag. We call this operation *product*. Removing the elements and concatenating all products is called *flattening*. For example, the product of

$$x[a[a_1],b[b_1],b[b_2]]$$

yields the following list of trees:

$$\mathrm{P}[a[a_1],b[b_1]],\mathrm{P}[a[a_1],b[b_2]]$$

The P-tags are generated by the product operator. The reason for wrapping each combination with an additional P-element is to ensure that elements that belong together stay grouped together. This is particularly important when further operations are applied. Note carefully that operator-generated tags will be ignored by matching, selection, projection, and all other functions that perform pattern matching. To summarize the above description with another example, the flattening of an element

$$\begin{aligned}r[&x[a[a_1],b[b_1],b[b_2]],\\ &y[a[a_2],a[a_3],c[c_1]]]\end{aligned}$$

yields the element:

$$\begin{aligned}r[&\mathrm{P}[a[a_1],b[b_1]],\\ &\mathrm{P}[a[a_1],b[b_2]],\\ &\mathrm{P}[a[a_2],c[c_1]],\\ &\mathrm{P}[a[a_3],c[c_1]]]\end{aligned}$$

Grouping

Grouping can be considered the inverse operation of flattening: a list of elements is transformed into a list of lists on the basis of common values for one (or more) sub-elements. Consider for instance, the following element:

$$t[e_1[a[1],b[2]],e_2[a[2]],e_3[a[1],c[9]]]$$

Grouping by a creates a group for each value of a which means to create an element with tag a and the value and a group element s that combines elements with a common value for a (the name s is typically a new name provided as an additional argument to the grouping operation). We obtain as a result the element

$$t[a[1], s[e_1[a[1], b[2]], e_3[a[1], c[9]]], a[2], s[e_2[a[2]]]]$$

More precisely, grouping (the sub-elements of) an XML value $t[x_1, \ldots, x_n]$ with respect to a tag u means to collect all those x_i into one group (that is, a list) whose sub-element of tag u (called *key*) have the same value, say, a. We refer to that sublist as xs_a. Hence, if $\{a_1, \ldots, a_n\}$ is the set of all different values for all the u-elements within the xs-elements, we obtain a partition of xs into different a_i-groups, that is, we have

$$xs =_{\{||\}} xs_{a_1}, \ldots, xs_{a_n}$$

where "$=_{\{||\}}$" denotes bag-equality, that is, equality of lists up to reordering. More precisely, grouping means to extract the common key (the u-element) from each a_i-group and to form a new element with tag G that contains the key followed by all the elements $x \in xs_{a_i}$ without the u-element. These "rest" elements are themselves wrapped into a tag R. Consider, for example, the value:

$$\begin{array}{l} x[t[a[a_1], b[b_1]], \\ \quad t[a[a_1], b[b_2]], \\ \quad t[a[a_2], b[b_2]], \\ \quad t[a[a_1]]] \end{array}$$

Grouping by tag a, respectively b, yields the following two elements:

$$\begin{array}{l} x[t[a[a_1], \\ \qquad \mathrm{G}[\mathrm{R}[b[b_1]], \mathrm{R}[b[b_2]]]], \\ \quad t[a[a_2], \\ \qquad \mathrm{G}[\mathrm{R}[b[b_2]]]]] \end{array} \qquad \begin{array}{l} x[t[\mathrm{G}[\mathrm{R}[a[a_1]]]], \\ \quad t[b[b_1], \\ \qquad \mathrm{G}[\mathrm{R}[a[a_1]]]], \\ \quad t[b[b_2], \\ \qquad \mathrm{G}[\mathrm{R}[a[a_1]], \mathrm{R}[a[a_2]]]]] \end{array}$$

We observe that in the second case we obtain three groups and that the first group contains only a "rest". This is because x contains an element $t[a[a_1]]$ that does *not* have a b-element, and for such "null"-values of keys, an additional group is constructed. It might seem that the operator-generated tags are not really needed, but in larger data examples containing many "t"-elements and where each "t"-element contains more sub-elements the need for grouping rests of "t"-elements as well as grouping all rest-elements for one group becomes necessary. We have described grouping for only one tag, but grouping can be reasonably performed for a set of tags. The formal definition given in the Appendix covers this more general case. It might also happen that an element contains two or more elements carrying the group tag. The formal definition takes care of this case, too.

4.2 Semantics of Patterns and Rules

We define the semantics of a Xing query by a mapping to basic semantic operations. In order to describe this mapping, we need an abstract syntax for document patterns and rules. We can use the tree notation introduced in Section 4.1 for this. Hence, a pattern is given by just a tree, and a rule is given by a pair of trees.

The semantics of single patterns is rather straightforward: just perform a selection with the pattern followed by a projection with the same pattern. Also the semantics of the extended patterns discussed in Section 3.3 fits nicely into this framework: for an extended pattern p, let $\overline{p}$ denote the pattern that is obtained from p by removing all marked sub-patterns. Then the semantics of p is given by the semantics of the rule $(p, \overline{p})$, that is, perform selection with p, and perform projection after that with $\overline{p}$.

More involved is the definition of rules since we have to explain how restructuring operations can be inferred from rules. For this we need to refer to the depth of a specific element in a tree, and we denote by $\delta(t;p)$ the depth of tag t in a pattern p.

Now when we process a result pattern we not just try to match the tags in the pattern to the (marked) value as was done by projection, in particular, we do not reject values on non-matching tags, but we rather try to identify the role of the non-matching tag. Since the selection was already performed by the argument pattern, non-matching tags encountered in the result pattern must have some special meaning. We can distinguish the following cases of non-matching tags. Let t be the non-matching tag of the result pattern r, and let y, ys be the list of elements to be processed. The argument pattern is p.

1. The tag is new, that is, the tag does not appear in the argument pattern. This is interpreted as a renaming on the root level and as an introduction of a grouping element on lower levels.
2. The tag in the result pattern is at a higher level than in the argument pattern. This is interpreted as a request for flattening data, and a flattening operation is applied $i = \delta(t;p) - \delta(t;r)$ times (that is, we compute $\varphi^i(y,ys)$, see Appendix) and then proceed with the current match.
3. The tag in the result pattern is at a deeper level than in the argument pattern. In that case there must be some tag u that is enclosing t and that is new in the result pattern, and there have to be also some tags ks that are at a higher level than t in r, but on the same level or lower as t in p. This is interpreted as a grouping operation (that is, the operation $\gamma(ks;y,ys;u)$ is performed before the matching continues, see Appendix).
4. The tag in the result pattern is at the same level as in the argument pattern, but at a deeper level than some tags ks that are on the same level as t in the argument pattern. In this case there must also be an enclosing tag u that is new in the result pattern. Again this is interpreted as a grouping operation, that is, the operation $\gamma(ks;y,ys;u)$ is performed, and the matching continues.

The third and the fourth case can always be identified by encountering the required new tag.

The semantics of Xing queries can be defined as follows (for a definition of μ, σ π, β, and ν, consult the Appendix).

$$\begin{aligned}
[\![p]\!](x) &= [\![(p,p)]\!](x) \\
[\![(p,r)]\!](x) &= \begin{cases} \pi(r;\sigma(p;x)) & \text{if } \mu(p;r) \\ \beta(r;\nu(r;\sigma(p;x))) & \text{otherwise} \end{cases}
\end{aligned}$$

The first equation expresses the fact that a single pattern is nothing but an abbreviation for a rule. The second equation distinguishes two cases based on whether argument and result pattern match. If they do match (μ), then the rule simply denotes a selection (σ) with the argument pattern followed by a projection (π) using the result pattern. If argument and result pattern do not match, the more involved value construction β is performed after the element has possibly be renamed by ν.

As an example, we demonstrate the semantics of the query shown in Figure 11. As alread explained earlier, the abstract representation of the query is given by the rule (p,r) where

$$\begin{aligned}
p &= \mathbf{bib}[*[\mathbf{title},\mathbf{author}]] \\
r &= \mathbf{byAuthor}[\mathbf{author},\mathbf{titles}[\mathbf{title}]]
\end{aligned}$$

Since p and r do not match, the semantics of $[\![(p,r)]\!](bib)$ is given by the second line of the second semantics equation. (Recall that *bib* is the XML value from Section 2.1, also visualized in Figure 1.) It is not difficult to see, that $\nu(r;\sigma(p;bib))$ yields the following tree (the effect of the renaming operation ν is just to change the tag of the root element form **bib** to **byAuthor**):

$$\begin{aligned}
\cdot\mathbf{byAuthor}[\cdot\mathbf{book}[&\text{year}[1988], \\
&\cdot\mathbf{title}[\text{Concrete Mathematics}], \\
&\cdot\mathbf{author}[\text{Graham}], \\
&\cdot\mathbf{author}[\text{Knuth}], \\
&\cdot\mathbf{author}[\text{Patashnik}]], \\
\cdot\mathbf{article}[&\text{year}[1998], \\
&\cdot\mathbf{title}[\text{Linear Probing and Graphs}], \\
&\cdot\mathbf{author}[\text{Knuth}], \\
&\mathbf{journal}[\text{Algorithmica}]]]
\end{aligned}$$

Next, β is applied to r and this intermediate value. The first line of the second equation for β is selected, and we get the expression:

$$\textbf{byAuthor}[\beta^*(\textbf{author}, \textbf{titles}[\textbf{title}]; \cdot\textbf{book}[\dots], \cdot\textbf{article}[\dots])]$$

In applying β^* we observe that x is **author** and $\tau(\textbf{author}) = \textbf{author}$. (Remember that the notation **author** as an element is an abbreviation for $\textbf{author}[()]$, so that strictly speaking, x is $\textbf{author}[()]$.) Since $\delta(\textbf{author}; p) = 2$ and $\delta(\textbf{author}; r) = 1$, we have to select the second line of the second equation for β^*, which means to apply the flattening operator to the list of the **book** and **article** element. We get as an intermediate result:

$$\begin{aligned}
&\text{P}[\cdot\textbf{title}[\text{Concrete Mathematics}], \cdot\textbf{author}[\text{Graham}]],\\
&\text{P}[\cdot\textbf{title}[\text{Concrete Mathematics}], \cdot\textbf{author}[\text{Knuth}]],\\
&\text{P}[\cdot\textbf{title}[\text{Concrete Mathematics}], \cdot\textbf{author}[\text{Patashnik}]],\\
&\text{P}[\cdot\textbf{title}[\text{Linear Probing and Graphs}], \cdot\textbf{author}[\text{Knuth}], \textbf{journal}[\text{Algorithmica}]]
\end{aligned}$$

The next iteration of β^* notices the new tag **titles** and performs a grouping operation which yields:

$$\begin{aligned}
&\text{P}[\cdot\textbf{author}[\text{Graham}],\\
&\quad\cdot\textbf{titles}[\text{R}[\cdot\textbf{title}[\text{Concrete Mathematics}]]]],\\
&\text{P}[\cdot\textbf{author}[\text{Knuth}],\\
&\quad\cdot\textbf{titles}[\text{R}[\cdot\textbf{title}[\text{Concrete Mathematics}]],\\
&\qquad\qquad\text{R}[\cdot\textbf{title}[\text{Linear Probing and Graphs}], \textbf{journal}[\text{Algorithmica}]]],\\
&\text{P}[\cdot\textbf{author}[\text{Patashnik}],\\
&\quad\cdot\textbf{titles}[\text{R}[\cdot\textbf{title}[\text{Concrete Mathematics}]]]]
\end{aligned}$$

Finally, β^* proceeds as a normal projection and yields:

$$\begin{aligned}
\textbf{byAuthor}[&\textbf{author}[\text{Graham}],\\
&\textbf{titles}[\textbf{title}[\text{Concrete Mathematics}]],\\
&\textbf{author}[\text{Knuth}],\\
&\textbf{titles}[\textbf{title}[\text{Concrete Mathematics}],\\
&\qquad\textbf{title}[\text{Linear Probing and Graphs}]],\\
&\textbf{author}[\text{Patashnik}],\\
&\textbf{titles}[\textbf{title}[\text{Concrete Mathematics}]]]]
\end{aligned}$$

Recall that all the generated tags are ignored in the matching process.

5 Advanced Patterns and Rules

In Sections 3.2 and 3.3 we have introduced the basic elements of the query system, patterns and rules, and we have seen how to express simple selection as well as more advanced data restructuring tasks. For many end users this might be already more than they are ever going to use, in particular, with regard to data restructuring. Nevertheless, there are situations in which the capabilities of the query interface shown so far are not sufficient to express certain queries. In this section we demonstrate some extensions to Xing that extend the expressiveness of the visual notation.

Or-Patterns

An *or-pattern* $P|Q$ consists of two distinct patterns P and Q and matches any pattern that is either matched by P or Q. Or-patterns are used to search for and combine data that might be stored in different elements. It allows a kind of generalization with respect to the schema of the data source. In its simplest form, an or-pattern is simply applied to tags, as for instance in **book**|**article**, which is actually identical to a regular expression tag pattern.

More interesting are or-patterns consisting of structured elements. For example, a search for publications from the JVLC or the VL-conference can be expressed by the or-pattern shown in Figure 13.

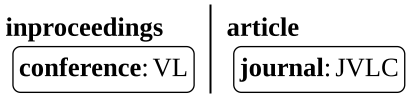

Figure 13: JVLC or VL publications

Like with regular expression patterns for tags, we also have to use an alias to refer to found elements in a result pattern. As an example, assume we look for "serious" publications where "serious" is defined to be either JACM articles or publications by Knuth. We can find the titles and authors of these publications by the following document rule:

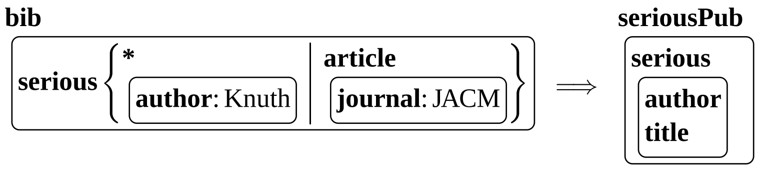

Figure 14: Serious publications

Quantification Patterns

Recall the queries asking for Knuth's publications: the first query (Figure 4) yielded too few information (only Knuth was listed as an author), and the second query (Figure 6) listed the complete bibliographic entries, which might be considered too much. What can we do if we wanted to see only the titles and *all* authors of publications by Knuth? One solution was suggested in Section 3.3: by using an extended form of patterns we can use sub-patterns only for selection and prevent them from performing projection, see Figure 8.

A more general solution is to introduce two special-purpose patterns: *universal* and *existential* list patterns. These are very handy in expressing conditions on repeated elements, such as authors. Whereas "**author**: Knuth" matches the list of author elements whose content equals "Knuth", the two patterns shown in Figure 15 match the whole list of authors if there is at least one author element matching "Knuth" (∃) or only if all author elements match "Knuth" (∀).

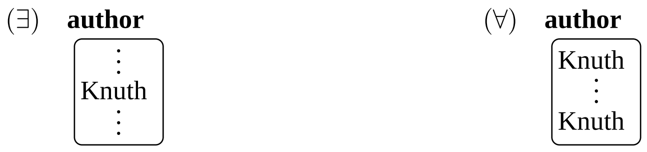

Figure 15: Universal and existential patterns

If the condition within an existential or universal pattern is itself just a text pattern (as in the above example), we again allow an abbreviating syntax (see Figure 16) that avoids one nesting level and is more readable, especially within larger contexts of somewhat complex queries.

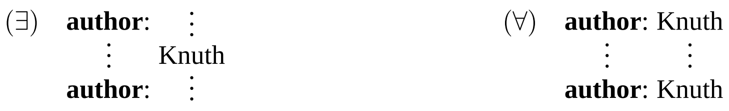

Figure 16: Abbreviations for universal and existential patterns

Hence, we can finally express the query asking for titles and authors of publications by Knuth as shown in Figure 17.

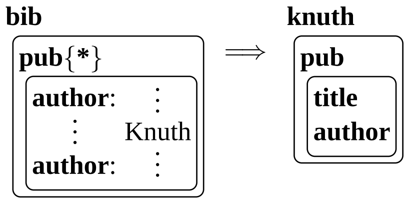

Figure 17: Titles and authors of Knuth's publications

Note that the universal pattern can be used to find all publications of which Knuth is the only author.

Deep Patterns

We can prefix any pattern *P* with an ellipsis "⋱" and obtain a deep version of that pattern, which matches *P* arbitrarily deeply nested in a document. This gives much more flexibility compared to using just *P* because *P* matches only elements on one document level. As an example consider the following deep pattern that retrieves all publication activities of Patashnik:

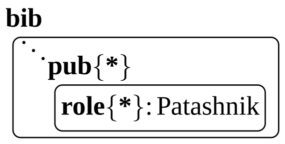

Figure 18: Publication efforts by Patashnik

This pattern is in fact generalizing in three ways: (i) it finds all kinds of bibliographic entries (articles, books, ...), (ii) it finds all kinds of elements having "Patashnik" as value (author, editor, ...), and (iii) it finds this information on any level, for example, publication entries that are nested within collection elements will be found as well as top-level entries.

Querying Multiple Data Sources

In database languages cartesian product and join operators are used to combine data from two or more different data sources. We can achieve the same effect by simply putting two or more patterns next to each other in a left-hand side of a rule. For example, assume that we have an XML document *person* containing information about people, such as affiliation or interests. Each entry is grouped into a **person** element which has the name of a person stored in a **name** element. Now we could ask, for instance, for all publications by authors from Stanford. This can be expressed as shown in Figure 19.

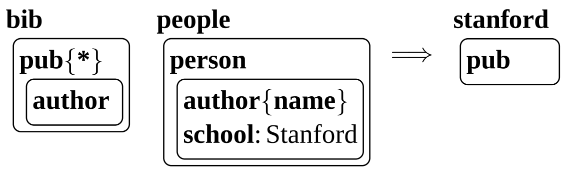

Figure 19: Publications by Stanford authors

Note how the join-condition is realized by using an alias pattern: the **name** element of *person* is renamed to **author**, and the use of equal variables signifies the condition that the bound values must be equal to become part of the result. If the names in the *person* document were, too, stored in **author** elements, then we would have not needed renaming and just could use the **author** tag.

By default, the join on list elements like **author** is defined with an existential semantics, that is, in the above example, all publication are found that have at least one author from Stanford. This is equivalent to using an existential pattern on the **author**-variable. In contrast, to find publications exclusively written by Stanford authors, we have to use a universal pattern. This is shown in Figure 20.

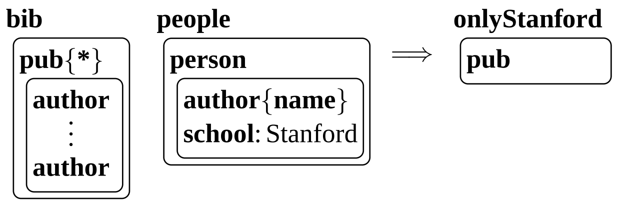

Figure 20: Publications exclusively by Stanford authors

Cross-Referencing

XML uses three special types of attributes to represent graph-like structures. Any element can have a single attribute of type ID. A value for such an attribute uniquely identifies that element. Other elements can refer to these identifiers through IDREF and IDREFS attributes, which establish links between the elements. This means, if an element *e* has a value “x17” for its ID-attribute, then any other element that has an IDREF attribute whose value is “x17” refers exactly to *e*. Whereas an IDREF attribute always contains a single reference, an IDREFS attribute contains a sequence of references and can thus establish links to multiple elements.

Continuing the previous example, assume each person element in *person* is defined to have an ID attribute called “personId”, and the author information in *bib* is realized as an IDREFS attribute called “authors”. Then the above join queries could still be posed value-oriented, that is, by matching equal values; we only have to change the element variables to attribute variables accordingly.

However, we can also use a graph-notation that is explicitly tied to ID and IDREF(S) attributes of elements: the fact that an attribute A references some element B is then represented as an arrow $A \rightarrow B$. The above example could then expressed as shown in Figure 21.

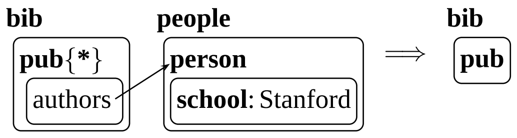

Figure 21: Publications by Stanford authors (referencing version)

Again, this query is interpreted with an existential default semantics. To find publication having only authors from Stanford we use a variation of the universal pattern for arrows, shown in Figure 22.

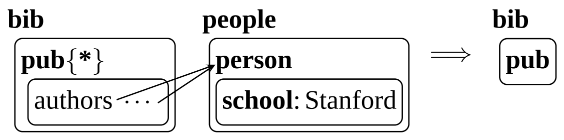

Figure 22: Publications exclusively by Stanford authors (referencing version)

Note that the use of cross-references makes it possible to build cyclic structures. If pattern matching is given the power to exploit them, visited elements have, in general, to be marked during pattern matching to ensure termination.

Aggregation

Dealing with collections of objects also requires some support for aggregating such collections. A standard approach is to introduce aggregation functions, such as *count*, and use expressions like *count*(**author**) in result patterns, or constrain queries by conditions like *count*(**author**) ≥ 2 (which could be put, for example, under the "$\Longrightarrow$" in rules).

Since there is no number data type in XML, aggregate functions like *maximum* or *average* cannot be easily defined. Hence, having actually only *count*-aggregation, we can well introduce a specialized visualization, at least with regard to conditions: put the condition next to the ellipsis of a universal pattern, see Figure 23.

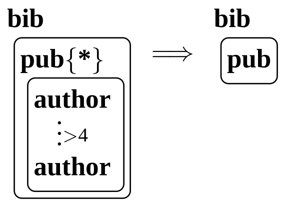

Figure 23: Finding publications with more than 4 authors

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 can be exploited in many different ways. One highly useful application is in the user interface. A user must somehow construct document patterns to build up XML queries. Typically, this happens through an editor that allows users to create boxes and to assign and change box tags. This might be tedious, in particular, for new users of a system because being faced with an empty screen without any clues can be frustrating and discouraging.

Now having a DTD available, creating named boxes can be made very easy. The first step for the user is to choose a document or database to query. This can be offered by a menu showing all XML resources known to the system. From this the user can select one (or even more). As an immediate feedback a pattern consisting just of a root element with the chosen name (or a regular expression denoting an alternative of names) is shown in the query space. In fact, for this very first step no DTD is needed at all, which is good since this means the first step of starting the construction of a query can be always performed without any additional requirements on the data resources.

After that the user might want to express conditions on the selected data source. What many users new to a system try if they do not have a clear idea of how to proceed is to click in menus or on objects available in the workspace. Now if we have a DTD for the selected data source, a mouse click on the root element can be used to pop up a menu showing all the possible attributes and sub-elements for the root element. From this menu one or more tags can be selected, and a box can be automatically inserted under the root element containing all the selected labels. This process can be repeated, and thus with a couple of mouse clicks, simple selection/projection queries can be quickly constructed.

By highlighting a tag before invoking the pop-up menu a different semantics can be expressed: in that case the selected tag is extended by the selections from the pop-up menu which means for the system to construct a regular expression tag (in this case, an alternative of element tags). By highlighting a tag together with its associated box before invoking the pop-up menu, an or-pattern can be constructed. As a special option each pop-up menu should contain a "*" so that queries asking for arbitrary sub-elements can be easily constructed.

In general, for novice users also tool tips support can be very helpful: for each currently pointed menu item a hint can be automatically displayed, like "select **book**s from **bib**" or for the wildcard "*": "select everything from **bib**".

There are certainly many more features conceivable that can be used to make the construction of queries comfortable for the user. But we believe that the possibility offered by DTDs to offer the user *context-dependent* support and hints (that is, which sub-elements are possible, etc.) is of utmost importance since it helps to quickly construct queries that are likely to return non-empty results. This encourages the user to further work with the system. Even in cases when no DTD exists, tools like XTRACT [25] can be used to generate DTDs that can then be employed at the user interface level.

7 Related Work

We have first proposed Xing in [18] and have emphasized user interface aspects in [19]. The present paper gives a more comprehensive description of the visual language and, in particular, provides a precise formal semantics for the core part. Other related work is described in the following subsections.

7.1 XML Query Languages

There exist quite a few query languages for XML by now. In all languages queries consist more or less of three parts for extracting, filtering, and formatting/restructuring data. The flavor of formulating queries differs, though, and there are also differences in their expressiveness:

XML-QL [16] uses, as we have already seen, XML patterns made out of tags, constant text and variables to denote parts of a document to be found. Construction of new elements can be freely mixed with query parts extracting desired elements. This makes XML-QL a very flexible approach. However, queries tend to become quite long, and one has to be comfortable in understanding XML-syntax to use it effectively. Lorel [26] uses a SQL-like language mainly extended by path expressions. Lorel allows users to express queries very succinctly, but it is difficult to create new elements or to rename elements, see Section 2.2. XQL [38] builds directly on XPath [11], which is a language for denoting parts of XML documents. This means, viewing XML values as trees, XPath expressions can be used to extract subtrees. XQL extends XPath by a number of operations, such as join or text containment. Although XQL can express joins, it is not possible to express regrouping as required by our third example query. Yatl [12] is similar to XML-QL: it also uses patterns to describe selections and constructions of elements. The syntax for patterns and expressions is not XML syntax, but similar to the succinct notation we have used in Section 4 and that can also be found in XDuce [27] or the most recent XML Query Algebra proposal [24]. An important difference in semantics is that Yatl preserves duplicates in flattening operations whereas XML-QL removes duplicates.

The XML query working group [34] has defined query requirements [8] (including a great number of different example applications and queries). Together with their definition of an XML query data model [21] and an XML query algebra [24, 22], this comprises a precise specification for XML query languages. The XML query data model is itself based on the XML information set [14] that provides a description of the information available in a well-formed XML document. The XML query algebra is intended to be a formal basis for an XML query language. The major parts are path expressions and a **for** construct that allows the formulation of iterations over sequences of elements. The XML query algebra proposal presents a very elegant and powerful core query system: together with conditional expressions and the ability to rather arbitrarily combine and nest different expressions selection, projection, join, quantification, and restructuring can be expressed. There is also support for sorting and aggregation. Quilt [9, 39] is a functional language that is similar to the XML query algebra, but provides specialized syntax for several algebra operations.

7.2 XML Processing in Programming Languages

Querying XML can be regarded as a special case of transforming XML, which can also be approached from a programming language point of view. In fact, there are also a number of proposals for providing special language support to write programs for processing XML. i The language supported by the W3C is XSLT [10], which is a rule-based language that is based on XPath. An important feature of XSLT-programs is that they are themselves XML documents so that they can be accessed, analyzed, transformed, etc., by any XML processing tool.

XDuce [27] is a functional language proposal that introduces the notion of regular expression types [29] and, in particular, offers regular expression pattern matching [28] that goes far beyond pattern matching found in languages like Haskell or ML and specifically accounts for locating data in XML documents.

In [44] two different approaches to perform XML processing in Haskell are proposed and compared: the first offers a set of combinators to express XML transformations whereas the second approach depends on the existence of a DTD and translates a DTD into corresponding Haskell types and functions on them.

7.3 Web Query Languages and Document Processing

There has also been work on the related issue of how to query the World-Wide Web. For example, W3QS [32] is a system offering a query language in the style of SQL that focuses on an extensible system architecture, and WebOQL [1] is a functional language that is based on powerful operators working on a versatile hypertree representation of arbitrary web contents. WebOQL's design is influenced by the underlying graph data model of UnQL [4] which was originally intended to be a query language for semi-structured data. All these approaches (and many others, for a survey, see [23]) view documents or web content as tree or graph structures and define textual query languages that are quite often similar to SQL.

In the area of document processing similar proposals have been made. Maruta [35, 36] describes a general tree model of documents and defines pattern matching by tree-regular languages and deterministic tree automata. His work generalizes the well-known approach of [31] by capabilities to express contextual conditions on document transformations.

7.4 Visual Languages

Since XML expressions are in essence multi-way trees, and thus it is apparent to denote these trees by pictures. XML-GL [7] is a language proposal that is based on exactly this idea. However, trees represent rather an abstract visual syntax of XML documents, which is well-suited for formal language manipulations [17] but not necessarily for end-user query languages. From this point of view, XML-GL is more like general-purpose visual programming languages that are based on trees and graphs, such as Progress [41] or Grrr [40]. An icon-based visual language for restructuring Web contents is presented in [37].

To achieve a high degree of usability, in particular, in the sense of *concreteness* [5] and *directness* [42], we instead propose to represent XML documents in a form-like way by nested rectangles with attached labels. On the one hand, this notation is related to the tree structure in a precisely determined way (namely by nesting) and thus directly reflects the XML objects that are represented.

A form-based query interface is also provided by EquiX [13] whose most important goal is to achieve a simple interface. However, the form metaphor is only used half-heartedly 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, and it is not even possible to reformat the query results—not to speak of data restructuring. It is not possible either to express conditions on arbitrarily deeply nested elements (which is possible in most textual XML query languages through path expressions or something similar). Such queries were termed *deep queries* in [4]. Similar constraints apply to QBE-like languages for the nested relational data model, for example, [33, 45]. Although these allow restructuring of data, they, too, depend on the presence of a schema and even demand the display of the complete schema of queried relations. They are not able to express deep queries either. DOODLE [15] and VQL [43] are form-based visual query languages that are intended as general-purpose database languages. Both are rule-based languages, and both are similar to Xing in their visual appearance using nested rectangles for displaying data as well as queries (although their screen space requirements are generally larger). In VQL, but not in DOODLE, it is possible to pose queries about the schema. This is a very important feature, especially in cases when the user has limited or even no knowledge of the schema. In Xing this is possible through the use of regular expressions as tags. Moreover, VQL is not capable of expressing deep queries. For a general survey of visual query languages, see [6].

8 Conclusions

We have presented a visual language to represent and query XML data. We believe that the language is immediately usable by a broad audience because of:

- the underlying document metaphor which reflects common knowledge about forms
- the readily accessible notion of document pattern, which relies, in particular, on implicit variables
- the absence of key words and the independence from a complex, textual formal query language

A prototype system is currently being implemented, and we intend to make it accessible to the public via the Internet. In particular, we are envisioning to ask interested users to construct sample queries to gather information about the usability. That information can be used to adapt and improve Xing.

We are also considering the extension of the document metaphor to a complete visualization of XPath. Based on this we can define various heterogeneous visual languages [20] for XML processing languages that built upon XPath, such as XSLT or Quilt.

Acknowledgments

The author thanks the anonymous reviewers for their very helpful comments.

References

[1] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring Documents, Databases and Webs. *Theory and Practice of Object Systems*, 5(3):127–141, 1999.

[2] A. Blackwell. See What You Need: Toward a Visual Perl for End Users. In *Visual End User Workshop*, 2000.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0*, 2000. `http://www.w3.org/TR/REC-xml`.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM SIGMOD Conf. on Management of Data*, pages 505–516, 1996.

[5] M. M. Burnett. Visual Programming. In Webster, J. G., editor, *Encyclopedia of Electrical and Electronics Engineering*, pages 215–283. John Wiley & Sons, 1999.

[6] T. Catarci, M. F. Costabile, S. Levialdi, and C. Batini. Visual Query Systems for Databases: A Survey. *Journal of Visual Languages and Computing*, 8:215–260, 1997.

[7] 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.

[8] D. Chamberlin, P. Frankhauser, M. Marchio, and J. Robie, editors. *XML Query Requirements*, 2000. `http://www.w3.org/TR/xmlquery-req`.

[9] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *3nd ACM SIGMOD Int. Workshop on The Web and Databases*, 2000.

[10] J. Clark, editor. *XSL Transformations (XSLT), Version 1.1*, 2000. `http://www.w3.org/TR/xslt11`.

[11] J. Clark and S. DeRose, editors. *XML Path Language (XPath), Version 1.0*, 1999. `http://www.w3.org/TR/xpath`.

[12] 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.

[13] 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.

[14] J. Cowan, editor. *XML Information Set*, 2000. `http://www.w3.org/TR/xml-infoset`.

[15] I. F. Cruz. DOODLE: A Visual Language for Object-Oriented Databases. In *ACM SIGMOD Conf. on Management of Data*, pages 71–80, 1992.

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

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

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

[19] M. Erwig. XML Queries and Transformations for End Users. In *XML 2000*, pages 259–269, 2000.

[20] M. Erwig and B. Meyer. Heterogeneous Visual Languages – Integrating Visual and Textual Programming. In *11th IEEE Symp. on Visual Languages*, pages 318–325, 1995.

[21] M. Fernández and J. Robie, editors. *XML Query Data Model*, 2000. `http://www.w3.org/TR/query-datamodel`.

[22] M. Fernández, J. Simeon, and P. Wadler. An Algebra for XML Query. In *Int. Conf. on Foundation of Software Technology and Theoretical Computer Science*, LNCS 1974, 2000.

[23] 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.

[24] P. Frankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler, editors. *The XML Query Algebra*, 2000. `http://www.w3.org/TR/query-algebra`.

[25] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *ACM SIGMOD Conf. on Management of Data*, pages 165–176, 2000.

[26] 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.

[27] H. Hosoya and B. C. Pierce. XDuce: A Typed XML Processing Language. In *3nd ACM SIGMOD Int. Workshop on The Web and Databases*, 2000.

[28] H. Hosoya and B. C. Pierce. Regular Expression Pattern Matching for XML. In *28th ACM Symp. on Principles of Programming Languages*, 2001.

[29] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *5th ACM Int. Conf. on Functional Programming*, pages 11–22, 2000.

[30] D. Jackson and M. Bell. String Pattern Matching in a Visual Programming Language. *Journal of Visual Languages and Computing*, 8(5/6):545–561, 1997.

[31] 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.

[32] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World Wide Web. In *21st Int. Conf. on Very Large Databases*, pages 54–65, 1995.

[33] N. A. Lorentzos and K. A. Dondis. Query by Example for Nested Tables. In *9th Int. Conf. on Database and Expert Systems Applications*, LNCS 1460, pages 716–725, 1998.

[34] M. Marchiori, editor. *XML Query*, 2000. `http://www.w3.org/XML/Query.html`.

[35] M. Maruta. Transformation of Documents and Schemas by Patterns and Contextual Conditions. In *3rd Int. Workshop on Principles of Document Processing*, LNCS 1293, pages 153–169, 1996.

[36] 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.

[37] M. Minas and L. Shklar. Visual Definition of Virtual Documents for the World-Wide Web. In *3rd Int. Workshop on Principles of Document Processing*, LNCS 1293, pages 183–195, 1996.

[38] J. Robie, editor. *XQL (XML Query Language)*, 1999. `http://www.ibiblio.org/xql/xql-proposal.html`.

[39] J. Robie, D. Chamberlin, and D. Florescu. Quilt: an XML Query Language. In *XML 2000*, pages 316–329, 2000.

[40] P.J. Rodgers. A Graph Rewriting Programming Language for Graph Drawing. In *14th IEEE Symp. on Visual Languages*, pages 32–39, 1998.

[41] A. Schürr, A. Winter, and A. Zündorf. Visual Programming with Graph Rewriting Systems. In *11th IEEE Symp. on Visual Languages*, pages 326–335, 1995.

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

[43] K. Vadaparty, Y. A. Aslandogan, and G. Ozsoyoglu. Towards a Unified Visual Database Access. In *ACM SIGMOD Conf. on Management of Data*, pages 357–366, 1993.

[44] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *4th ACM Int. Conf. on Functional Programming*, pages 148–159, 1999.

[45] L. Wegner, S. Thelemann, S. Wilke, and R. Lievaart. QBE-like Queries and Multimedia Extensions in a Nested Relational DBMS. In *Int. Conf. on Visual Information Systems*, pages 437–446, 1996.

Appendix

In the semantics definitions below we use single letters, such as $x, y, \ldots$ to denote single elements and strings ending with an "s" to denote lists of elements, for example, $xs, ys, \ldots$. This helps to identify the type of variables through their name.

Matching

The definition of matching is by induction on the structure of patterns and traverses the pattern and the value in parallel: μ determines whether two elements match, and μ^* checks whether two lists of elements match. For matching tags, μ uses a predicate μ_S that performs ordinary string matching and that is assumed to be defined elsewhere. In the following we use the semicolon also to separate function arguments.

$$\begin{aligned}
\mu(t\,[\,()\,]; u\,[\,()\,]) &= \mu_S(t;u) \\
\mu(t\,[xs]; u\,[ys]) &= \mu_S(t;u) \wedge \mu^*(xs;ys)
\end{aligned}$$

$$\begin{aligned}
\mu^*((); ()) &= \mathit{false} \\
\mu^*(x,xs;y,ys) &= \mu(x;y) \vee \mu^*(xs;ys)
\end{aligned}$$

This definition says that for a pattern to match an element both tags must (string-) match and, if existent, at least one sub-pattern must match a corresponding sub-element.

Selection

As done for matching, the definition of selection is performed by induction on the structure of patterns and uses two mutually recursive functions σ and σ* that traverse the pattern and the value in parallel. The definition of σ is rather simple since the actual work of pattern matching is done by μ and μ^*.

$$\sigma(t\,[xs]; u\,[ys]) = \begin{cases} \cdot u\,[\sigma^*(xs;ys)] & \text{if } \mu(t\,[xs]; u\,[ys]) \\ u\,[ys] & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\sigma^*(xs; ()) &= () \\
\sigma^*(x; y, ys) &= \begin{cases} \cdot y, \sigma^*(x;ys) & \text{if } \mu(x;y) \\ \sigma^*(x;ys) & \text{otherwise} \end{cases} \\
\sigma^*(x, x', xs; y, ys) &= \begin{cases} \cdot y, \sigma^*(x, x', xs; ys) & \text{if } \mu(x;y) \\ \cdot y, \sigma^*(x', xs, x; ys) & \text{if } \neg\mu(x;y) \wedge \mu(x';y) \\ \sigma^*(x, x', xs; ys) & \text{otherwise} \end{cases}
\end{aligned}$$

Note in the last equation how the first pattern x is appended to the end of the pattern list whenever the second pattern x' matches; this is to achieve the iterative matching behavior.

Projection

The unmark function removes marks from final results; it is defined as follows:

$$\begin{aligned}
\upsilon(\cdot t\,[xs]) &= t\,[\upsilon^*(xs)] \\
\upsilon(t\,[xs]) &= t\,[\upsilon^*(xs)]
\end{aligned}$$

$$\begin{aligned}
\upsilon^*(()) &= () \\
\upsilon^*(x, xs) &= \upsilon(x), \upsilon^*(xs)
\end{aligned}$$

In the following definition of projection the final cases express that projection on unmarked values yields nothing, irrespective of the used pattern.

$$
\begin{array}{lcl}
\pi(t[()];\cdot u[ys]) & = & \begin{cases} u[\upsilon^*(ys)] & \text{if } \mu_S(t;u) \\ () & \text{otherwise} \end{cases} \\
\pi(t[xs];\cdot u[ys]) & = & \begin{cases} u[\pi^*(xs;ys)] & \text{if } \mu_S(t;u) \\ () & \text{otherwise} \end{cases} \\
\pi(x;y) & = & ()
\end{array}
$$

$$
\begin{array}{lcl}
\pi^*(xs;()) & = & () \\
\pi^*(x,x',xs;\cdot y,ys) & = & \begin{cases} \pi(x;y),\pi^*(x,x',xs;ys) & \text{if } \mu(x;y) \\ \pi(x';y),\pi^*(x',xs,x;ys) & \text{if } \neg\mu(x;y)\wedge\mu(x';y) \\ \pi^*(x,x',xs;ys) & \text{otherwise} \end{cases} \\
\pi^*(x;\cdot y,ys) & = & \begin{cases} \pi(x;y),\pi^*(x;ys) & \text{if } \mu(x;y) \\ \pi^*(x;ys) & \text{otherwise} \end{cases} \\
\pi^*(xs;y,ys) & = & \pi^*(xs;ys)
\end{array}
$$

Flattening

Flattening an XML value means to eliminate a level of its hierarchy.

Before we can define the product and flattening operations, we need some auxiliary definitions and notations: τ yields the tag of an element, and θ gives the list of children for an element. Applied to a list of elements, τ yields the list of all elements' tags.

$$
\begin{array}{lcl}
\tau(t[xs]) & = & t \\
\tau(xs) & = & (\tau(x) \mid x \in xs) \\
\theta(t[xs]) & = & xs
\end{array}
$$

We use $(e(x) \mid x \in xs \wedge p(x))$ to denote the list of $e(x)$ for those xs for which the property p holds. If p is missing, it denotes the list obtained by applying e to all $x \in xs$. We may also use more than one iterator set, for example, $(e(x,y) \mid x \in xs, y \in ys)$ denotes the list of expressions obtained by ranging over all combinations of xs and ys (ys make up the "inner" loop and may depend on x) . Finally, to aggregate the elements of a list xs with a binary operation f we write f/xs which denotes the expression $f(\ldots f(f(x_1,x_2),x_3),\ldots,x_n)$, for example, $+/(x_1,x_2,x_3) = (x_1+x_2)+x_3$.

Now we can define the function ω that yields for each sub-element's tag a list of elements with this tag; in a sense, it produces a kind of relational view for an XML element by computing a list of columns where each column is a list of elements of the same tag. For example, we have

$$\omega(x[a[a_1],b[b_1],b[b_2]]) \;=\; a[a_1];b[b_1],b[b_2]$$

The definition for ω is:

$$\omega(t[xs]) \;=\; ((y \mid y \in xs \wedge \tau(y) = u) \mid u \in \tau(xs))$$

Next we need a "right product" operator $\overleftarrow{\times}$ that takes a list of element lists $xs_1;\ldots;xs_n$ and a list ys and extends each xs_i by each $y \in ys$. (The name "right product" indicates that the extension of the element lists happens at the right end.) $\overleftarrow{\times}$ will be repeatedly applied to all the columns delivered by ω to build up element combinations (which can be thought of as "tuples" in the sense of relational databases) tag by tag. Hence, if each argument list xs_i is a column, each intermediate result of an application of $\overleftarrow{\times}$ represents a kind of list of "partial" tuples containing the columns processed thus far.

$$ts \overleftarrow{\times} ys \;=\; (xs,y \mid xs \in ts, y \in ys)$$

With $\overleftarrow{\times}$ we can define the operation ρ that computes the product of an element: we simply apply $\overleftarrow{\times}$ to all columns of that element. In order to retain the tuple structure we additionally have to wrap each tuple in a special tag P. Finally, the flattening operation φ applies ρ to all children of an element and concatenates the resulting lists. Sometimes flattening has to be applied directly to a list of elements and not to a tree. We also supply a definition for this case.

$$\begin{aligned}
\rho(t[xs]) &= (\mathrm{P}[y] \mid y \in \overleftarrow{\times}/\omega(t[xs])) \\
\varphi(t[xs]) &= t[\varphi(xs)] \\
\varphi(xs) &= (y \mid x \in xs, y \in \rho(x))
\end{aligned}$$

Grouping

To define the grouping operation formally we need some auxiliary definitions. First, we have an operation ζ that partitions the children of an element into two lists according to a given set of tags that specify the key. For example,

$$\zeta(a; t[a[a_1], b[b_2], a[a_2], c[c_1]]) = t[\mathrm{K}[a[a_1], a[a_2]], \mathrm{R}[b[b_2], c[c_1]]]$$

ζ is defined by:

$$\zeta(ts; t[xs]) = t[\mathrm{K}[(x \mid x \in xs \wedge \tau(x) \in ts)], \mathrm{R}[(x \mid x \in xs \wedge \tau(x) \notin ts)]]$$

Once the keys are exposed in all elements, the elements have to be arranged into groups of equal keys. This is done by the operation κ, which is defined below. In the definition we use the function #1 that selects the first element from a list.

$$\begin{aligned}
\kappa(()) &= () \\
\kappa(t[k,r], xs) &= t[k,r], ys; \kappa(zs) \quad \text{where} \quad ys = (x \mid x \in xs \wedge \#1(\theta(x)) = k) \\
&\qquad\qquad\qquad\qquad\qquad\qquad zs = (x \mid x \in xs \wedge \#1(\theta(x)) \neq k)
\end{aligned}$$

The final step is to extract the key from each list obtained by κ and wrap the key and the list of rest elements into a separate group element. This is done by the function ξ that accepts a list of trees (which all have the same key) and computes a single element that has the tag of the first element in the list and contains the key elements (without the operator-generated wrapping tag K) followed by a group element (with a tag that can be supplied as an additional function argument) that contains the list of rest elements.

$$\xi(t[k,r], xs; u) = t[\theta(k), u[r, (r' \mid t[k, r'] \in xs)]]$$

Now grouping of an element $t[xs]$ with respect to a set of key tags ts can be essentially performed in four steps:

1. Sort xs. (Sorting is performed on tags and, on equal tags, recursively by sub-elements.)
2. Partition all elements $x \in xs$ according to ts.
3. Identify sublists xs_{a_i} of equal key-elements.
4. Extract key-elements from each each xs_{a_i}.

These steps lead finally to the following definition for γ. Note that γ takes three arguments: (i) the tags for the key elements on which to perform grouping, (ii) the tag to be used for the group element, and (iii) the element to be grouped. As with flattening, γ can be also applied to a list of elements. We include a definition for this case, too.

$$\begin{aligned}
\gamma(ts; u; t[xs]) &= t[\gamma(ts; u; xs)] \\
\gamma(ts; u; xs) &= \xi(\kappa((\zeta(ts; x) \mid x \in sort(xs))); u)
\end{aligned}$$

Inferring Restructuring Operations

The depth of a tag in a pattern can be computed by the function δ that searches for the first occurrence of the tag:

$$\delta(u; t[xs]) = \begin{cases} 0 & \text{if } t = u \\ 1 + \min\{\delta(u; x) \mid x \in xs\} & \text{otherwise} \end{cases}$$

If a tag does not occur in a tree, its depth is defined to be $\perp$.

We determine the tags of the key and the rest elements with the functions τ_K and τ_R and pass the found tags to the γ operation. We define that γ inserts new group tags with marks so that they are not ignored during the further building process. When x is the element whose tag is new, say $\tau(x) = \textbf{titles}$ as in Figure 11, the tags for the rest elements that appear inside the group element are directly given by the tags of x's children, in this case **title**. The key tags can be obtained either (i) by determining in the pattern p the siblings of $\tau(\theta(x))$-elements or, if there are none, (ii) by taking the children tags of that element in p that has the tag $\#1(\tau(\theta(x)))$. To express this selection it is useful to have a further function ϕ_T that retrieves the children of the first sub-element in a tree having a specific tag and a function ϕ_C that finds a list of elements (that are siblings) that contain a set of specified tags.

$$\begin{aligned}
\phi_T(t;x) &= \begin{cases} \theta(x) & \text{if } \tau(x) = t \\ \theta(\#1(\phi_T(t;y) \mid y \in \theta(x))) & \text{otherwise} \end{cases} \\
\phi_C(ts;x) &= \begin{cases} \theta(x) & \text{if } (\tau(y) \mid y \in \theta(x)) \supseteq ts \\ \theta(\#1(\phi_C(ts;y) \mid y \in \theta(x))) & \text{otherwise} \end{cases} \\
\tau_K(x) &= \begin{cases} ks & \text{if } ks \neq () \\ (\tau(y) \mid y \in \phi_T(\#1(r);p)) & \text{otherwise} \end{cases} \\
& \quad \text{where} \quad r = \tau(\theta(x)) \\
& \qquad\qquad ks = (\tau(y) \mid y \in \phi_C(r;p) \wedge \tau(y) \notin r)
\end{aligned}$$

This leads to the definition of the functions β and β^* that build values with a result pattern.

$$\begin{aligned}
\beta(t[()];\cdot u[ys]) &= \begin{cases} u[\upsilon(ys)] & \text{if } \mu_S(t;u) \\ () & \text{otherwise} \end{cases} \\
\beta(t[xs];\cdot u[ys]) &= \begin{cases} u[\beta^*(xs;ys)] & \text{if } \mu_S(t;u) \\ () & \text{otherwise} \end{cases} \\
\beta(x;y) &= ()
\end{aligned}$$

$$\begin{aligned}
\beta^*(xs;()) &= () \\
\beta^*(x,x',xs;\cdot y,ys) &= \begin{cases} \beta(x;y), \beta^*(x,x',xs;ys) & \text{if } \mu(x;y) \\ \beta^*(x,x',xs;\varphi^k(\cdot y,ys)) & \text{if } \exists\, z \in x,x',xs : k = \delta(\tau(z);p) - \delta(\tau(z);r) > 0 \\ \beta^*(x,x',xs;\gamma(\tau_K(z);\cdot y,ys;\tau(z))) & \text{if } \exists\, z \in x,x',xs : \delta(\tau(z);p) = \perp \\ \beta(x';y), \beta^*(x',xs,x;ys) & \text{if } \mu(x';y) \\ \beta^*(x,x',xs;ys) & \text{otherwise} \end{cases} \\
\beta^*(x;\cdot y,ys) &= \begin{cases} \beta(x;y), \beta^*(x;ys) & \text{if } \mu(x;y) \\ \beta^*(x;ys) & \text{otherwise} \end{cases} \\
\beta^*(xs;y,ys) &= \beta^*(xs;ys)
\end{aligned}$$

The above definition is similar to that of π but investigates the case for non-matching tags more deeply. As a side condition we require that for each tag a flattening operation is performed at most once.

In the definition for β we have omitted the special case for checking the root element for renaming. It is easier to check this and perform a possible renaming before entering the function β. We use the function ν for this:

$$\nu(t[xs];\cdot u[ys]) = \begin{cases} \cdot t[ys] & \text{if } t \neq u \\ \cdot u[ys] & \text{otherwise} \end{cases}$$

The semantics of Xing queries can now be defined as follows.

$$\begin{aligned}
[\![p]\!](x) &= [\![(p,p)]\!](x) \\
[\![(p,r)]\!](x) &= \begin{cases} \pi(r;\sigma(p;x)) & \text{if } \mu(p;r) \\ \beta(r;\nu(r;\sigma(p;x))) & \text{otherwise} \end{cases}
\end{aligned}$$