A Visual Language for XML

Martin Erwig
Oregon State University
erwig@cs.orst.edu

Abstract

XML is becoming one of the most influential standards concerning data exchange and Web-presentations. In this paper 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. It combines an intuitive, dynamic form-based interface for defining queries and transformation rules with powerful pattern matching capabilities and offers thus a highly expressive yet easy to use visual language.

Providing visual language support for XML not only helps end users, it is also a big opportunity for the VL community to receive greater attention.

1 Introduction

The eXtensible Markup Language (XML) is a standardized notation for documents and other data [3]. It is very likely that XML develops into the prevailing format for exchanging documents and presenting data in the World-Wide Web. An XML document contains, in addition to its content, information about its structure, which is achieved through tagged 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 `</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="1998">`.

If all begin and end tags are properly nested in an XML document, the document is said to be *well-formed*, and it describes essentially a tree. This structure can be constrained further by a DTD (Document Type Definition), which can be thought of as a grammar defining attributes and possible nestings of elements. A DTD represents a kind of schema or type information for XML documents. An XML document conforming to a DTD is said to be *valid*.

We do not consider DTDs in the first place 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 language to have knowledge of the concept of a "schema". Nevertheless, there are ways to nicely exploit DTDs when they are given. We shall discuss this issue in Section 5.

Let us consider a small example. Below we show some bibliographic data in XML format.

```
<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 refer to this XML value as *bib*. A typical query on such a data is, for example, to find all publications of a particular author; this leaves the structure of the queried data resource essentially unchanged. In contrast, the task of building, for example, a list of authors with the titles of their publications is an example that needs restructuring of data.

In the following we present a visual language, called *Xing* (pronounced "crossing" and abbreviating **X**ML **in** **g**raphics), that allows to denote XML data, queries, and programs in a way that facilitates the comprehension of the data's internal structure. To give an impression of Xing, the representation of the above bibliography data as a Xing document is shown in Figure 1.

Elements are depicted as boxes with the element's tag written above it. Attributes and subelements are displayed within the border of the father element. Subelement tags are distinguished from attribute names by using boldface type. The order of the subelements is given by their relative vertical position. If a subelement contains no attributes and only text, it can also be displayed simply in a textual way as a tag, followed by a colon and its value. This additional rule allows to simplify in many cases the visual appearance.

So far this is hardly exciting, but the notation gets really

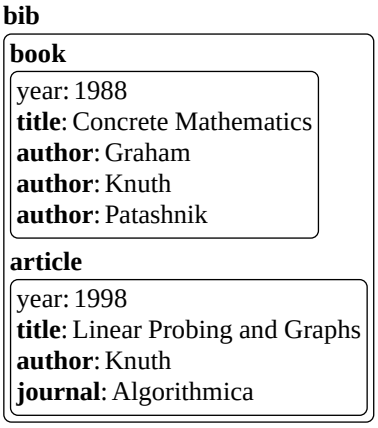
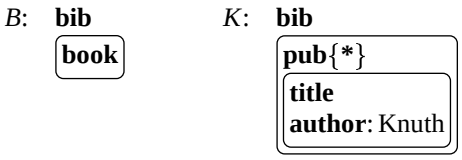


Figure 1. Bibliography as a Xing expression

useful when formulating XML queries and programs. In its simplest form, a query is expressed by a *document pattern*, which is much like an expression, except that elements tags and attribute names are used as variables. For example, below are two patterns/queries for finding all books and all publications of Knuth:



Such queries are evaluated by inductively matching patterns against XML data and returning the matched subdocument. For example, when matching *B* against *bib*, the tags of both outermost elements match, and the tag **book**, which is used here simply as a variable, matches the first of the two bibliographic entries and binds it to **book**. The result is the bibliography with just the book entry. The second pattern matches both entries by using a regular expression * as a tag (and assigning an additional tag name **pub** to any found element). For any subelement, **title** matches because this element tag is used here simply as a variable. In contrast, only those **author**-elements that have the value "Knuth" match. In this example the resulting document contains parts of both publication entries, see Figure 2.[1]

In the rest of this paper we will discuss related work in Section 2, and we comment upon the importance of XML in Section 3. In particular, what can XML and visual languages contribute to each other. We present the basic visual language for querying and restructuring XML in detail in Section 4. More advanced features, such as expressing joins and dealing with object identity and cross references

[1]We will explain the details of the pattern matching and query evaluation process below, for example, why is the year information shown in the result even though no such variable was specified in the query, or why is only Knuth shown as an author of the book, or why isn't the journal shown, and how can we change that, ...

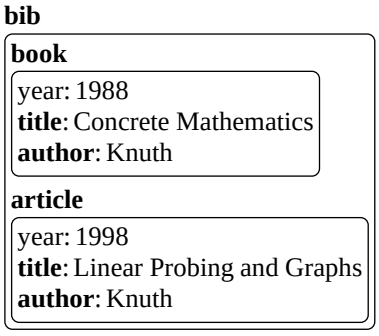


Figure 2. Result of Query *K*

are briefly discussed in Section 5; there we will also comment upon exploiting DTDs. Conclusions and remarks on future work are finally given in Section 6.

2 Related Work

There exist quite a few query languages for XML by now, for example, XML-QL [11], Lorel [14], YatL [8], and XQL [22]. 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. For instance, XML-QL uses XML patterns made out of tags, constant text and variables to denote parts of a document to be found, Lorel uses a SQL-like language, and XQL mainly supports navigation by regular expressions. All these languages have in common that they are text-based.

There has, of course, also been work on the related issue of how to query the World-Wide Web. For example, W3QS [16] 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 [13]) 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 [18, 19] 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 [15] by capabilities to express contextual conditions on document transformations. Approaches to text databases frequently trade expressiveness for efficiency [2].

We have already noted that 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 [12] 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 [24] or Grrr [23]. An icon-based visual language for restructuring Web contents is presented in [21].

To achieve a high degree of usability, in particular, in the sense of *concreteness* [5] and *directness* [25], 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. On the other hand, it is much easier to deal with from the user's point of view.

A form-based query interface is also provided by EquiX [9] 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, [17, 27]. 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 [10] and VQL [26] 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. However, neither VQL nor DOODLE is capable of expressing deep queries. For a survey of visual query languages, see [6].

3 XML and Visual Languages

XML is developing into a standard markup language for documents, in particular, for documents to be presented on the World-Wide Web. Since XML simplifies many aspects of the more complex SGML, it can be expected to be more widely accepted and to receive vital support through tools etc. Moreover, XML is also recognized as a standard format for exchanging data, and there is a rapidly growing number of XML applications. Many large and influential computer companies invest into XML technology and have announced to support XML.

Hence, it can be expected that the vast amount of tomorrow's data and web resources are available in XML format; a large part of it will be even available exclusively in XML format.

Thus, there is a very strong demand for languages for processing XML documents and data resources. In particular, querying and transforming XML data from one format into another will be an omnipresent everyday's task. The fact that almost every computer user is, via the Internet, a potential user of XML data presents a challenge to design powerful yet easy to use languages for processing XML. At the same time this is a big chance for visual languages to reach a larger audience.

3.1 What Can Visual Languages Contribute to XML?

Text is good for representing linear information, but it is tedious to retain or identify structure in textual representations of hierarchical or even graph-like organized informations. Often explicit layout, such as indenting or bracketing, is used to emphasize the underlying structure. Hence, XML documents are predestined to visual representation. The same is true for languages to extract information from XML documents or to restructure XML data because querying essentially means to identify values and structures in documents, and this can be conveniently done visually. Transformation of XML data can be described by combining querying (via patterns) and constructing result values, which again is suited very well for visual notation.

3.2 What Has XML to Contribute to Visual Languages?

During last year's VL conference, UML has been identified as one "killer application" for visual languages, that is, many people believe that when the visual language community can come up with semantics/tools/... for UML, visual languages themselves will be recognized by more people and will enjoy a much wider acceptance. These goals are, however, quite difficult to achieve, in particular, since UML is not very much formalized yet. This is aggravated by the fact that UML is a collection of several different notations, and it is far from being clear how these can be integrated (on a formal level).

In contrast, XML has a rather simple structure, hence providing models, semantics, and tools can be done much easier and faster. Therefore, we believe that XML is even

more suited to be a "killer application" for visual languages. And the impact of visual language technology would be even greater because XML is expected to be just about everywhere.

Hence, we believe that XML should be a major theme of visual language research, this paper being one contribution.

4 Xing

In this section we describe the visual query and restructuring language in more detail. A Xing program is essentially given by a rule that is itself made out of two patterns. A pattern in a left-hand side of a rule creates bindings that are used by the pattern on the right-hand side to construct new documents.

We start in Section 4.1 with a brief sketch of the document metaphor that forms the basis of the design of Xing. Then in Section 4.2 we define a model of nested bindings and describe the different kinds of patterns that are available in Xing. Rules and simple queries are introduced in Section 4.3, and language features to facilitate more complex data restructuring tasks are presented in Section 4.4.

4.1 The Document Metaphor

Many of the documents we are used to dealing with in everyday's life (faxes, product descriptions, all kinds of forms, etc.) typically contain more or less portions of free text together with categorized informations (fields). Fields consist of a *header*, that is, a short textual description, and a *value*, which is given by text or/and another structured part, for example, a collection of further fields.

Our visual representation mimics exactly these kinds of documents which are well-known to most people. This is especially supported by the abbreviating notation for unstructured elements, such as **author**: Knuth. Therefore, the visual language should be easily accessible even to novice computer users. This is also strongly supported by the fact that simple queries (that is, document patterns) are just sketches of documents containing only material that is also used in documents, and these can be expressed with just the knowledge of what constitutes a document. In particular, no concept of things like keyword, variable, etc., is needed on the part of the user. This is because tags are implicitly interpreted as variables, text constants work as selections, and the act of mentioning or omitting tags is additionally interpreted as projection of data.

Of course, to pose more advanced queries, in particular, to perform data restructuring, some abstract concepts of Xing, such as patterns or rules, have to be realized, but as we show in the following subsection, there are only few simple syntactic rules that have to be learned.

4.2 Patterns and Bindings

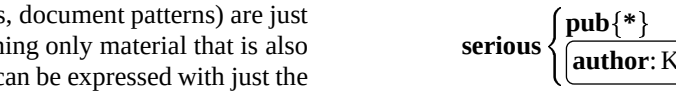
The usual notion of a binding is a mapping from identifiers to some kind of values. We refine this by saying that a *nested binding* is a mapping from identifiers to a sequence of objects where an object is either a string value, an attribute, an element, or a nested binding itself.

In the following we describe the different kinds of patterns together with their behavior with regard to matching and binding.

Text Patterns. These are regular expressions over string values, that is, constants, such as "Knuth" or "1998", and expressions like "(Con|Dis)crete Math*". We use the convention that a * not following a bracket is a shorthand for ".*", otherwise, regular expressions are defined to match string values in the usual way. If matching succeeds, a corresponding binding for the pattern and its matching string is produced.

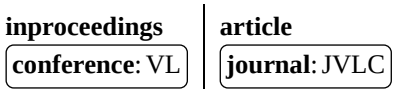
Name Patterns. They come in four different versions:

1. An *attribute pattern* is given by a text pattern, such as "year", and it matches and binds only attributes whose name matches the pattern.
2. A *tag pattern* is also given by a text pattern, for example, **book** or **book**|**article**, and is matched against a list of elements. If the sublist of elements that have a matching tag is not empty, the match succeeds and produces a binding for that list.
3. An *alias pattern* consists of a tag name, such as **new** and a pattern P and is written as **new**$\{P\}$. It matches in exactly the same way as P does, but allows to refer to the resulting bindings also through **new**. It is useful in connection with or-patterns (see below) and regular expressions used in tag patterns. Note that an alias can be also introduced for complex patterns. For example, if we look for "serious" publications where serious is defined to be either JACM articles or publications by Knuth, we can find all these items and collect them under the tag **serious** by using the following alias pattern:



4. A *variable pattern* like *title* matches and binds either attributes or elements of the same name. It is very useful when the schema of the queried data is not known. Then one can use a pattern *title* and be sure to find titles in bibliographic databases irrespective of whether titles are realized as attributes or as elements.

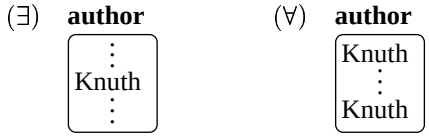
Or-Patterns. These are used to search for and combine data that might be stored in different elements. It allows a kind of generalization with respect to schema of the data source and can be applied to tag patterns, as in **book**|**article** (this is identical to a regular expression tag pattern), or even to nested patterns (described next). For example, a search for publications from the JVLC or the VL-conference can be expressed by the following or-pattern:



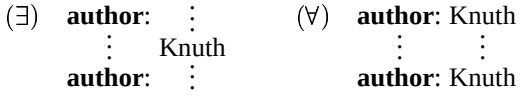
Another example has been already given with the alias pattern **serious**. Or-patterns match all values that match any of the individual patterns contained in the or-pattern. No new, separate binding is created, however. To combine the bindings of the contained patterns, an alias pattern can be used.

Nested Patterns. Such a pattern consists of a header, which is a tag pattern, and a body, which is given by a non-empty sequence of (arbitrary) patterns. For example, "**author**: Knuth" is a nested pattern, and so are the patterns B and K from the Introduction. A nested pattern P with header H and body $B = P_1, \ldots, P_n$ matches all elements that (i) have a tag that matches H and (ii) that contain attributes $a_1, \ldots, a_l$ and subelements $e_1, \ldots, e_m$ such that there is a subset of attributes and a subsequence of elements that match B. This means that there is a $k \in \{0, \ldots, n\}$ and two mappings $\alpha : \{1, \ldots, k\} \to \{1, \ldots, l\}$ and $\varepsilon : \{1, \ldots, m\} \to \{k+1, \ldots, n\}$, such that P_i matches $a_{\alpha(i)}$ for $1 \leq i \leq k$ and P_i matches the list $[e_u, \ldots, e_v]$ for $k < i \leq n$ where $\{u, \ldots, v\} = \varepsilon^{-1}(i)$. As a result, a binding for the list of all matching elements is produced, and all subbindings resulting from matching the subpatterns are included, too.

Two special-purpose nested patterns are *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 following two patterns match the whole list of authors if there is at least one author element matching "Knuth" ($\exists$) or only if all author elements match "Knuth" ($\forall$).

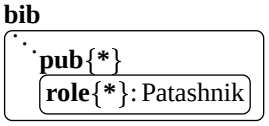


If the condition within an existential/universal pattern is itself just a text pattern, we again allow an abbreviating syntax that avoids one nesting level and is more readable, especially within larger contexts of somewhat complex queries:



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 name, or-, or nested pattern P with an ellipsis "$\ddots$" 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:



This pattern is 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.

More examples for the different kinds of patterns are presented in the following subsections.

4.3 Rules and Basic Queries

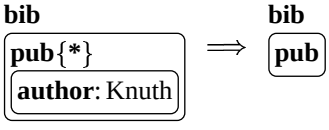
The most simple form of a query is given just by a document pattern – like B and K from the Introduction.

In fact, any such a pattern used as a query is only an abbreviation for a *document rule* which has the form $P \Rightarrow R$ where P and R are both patterns. P is called the *argument pattern*, and R is called the *result pattern*. Now, a single pattern like B is just a shorthand for the rule $B \Rightarrow B$.

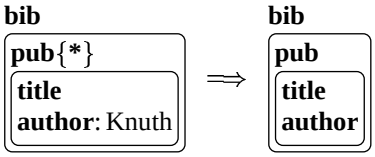
Next we can explain several aspects concerning the result for the query K from the Introduction:

(1) We have defined the query semantics to include, by default, all attributes because attributes are more closely tied to elements than subelements are. Indeed, any atomic subelement can as well be modeled as an attribute and vice versa. Now it can be expected that attributes will be used for regular data whereas elements are preferred for more variable data. However, there are no design guidelines for XML or DTDs available yet, and this issue can be considered open. We can always suppress the binding and display of all attributes by putting a "-" after the element tag. Selecting some of the attributes can be achieved simply by using corresponding attribute patterns in the body of a nested pattern.

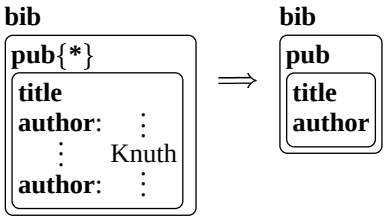
(2) A variable is always bound to a complete element, and if this variable is used without further constraints (that is, without showing subelements), this element is shown completely in the result. Hence, **pub** in K is indeed bound to both complete bibliographic elements. However, if an element of a result pattern, contains subelements, these have the effect of projecting only the bindings of the corresponding subelements. This is the reason why **journal** does not appear in the article element of the result and Knuth's coauthors do not appear in the result book entry. Hence, to show the complete bibliographic entries we can simply use a separate result pattern using just the variable **pub**.



Now what can we do if we wanted to know only the titles and authors? We could be tempted to try the rule:



However, this does not work because the pattern binds only "Knuth" **author**-elements. What we need instead is the existential pattern ($\exists$) from above:



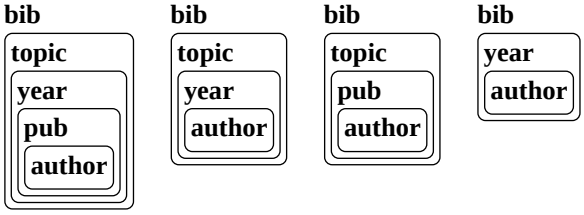
There are many other variations conceivable, but before we consider more queries we shall describe in more detail how a document is constructed from a right-hand side pattern and a binding.

4.4 Data Restructuring by Grouping and Ungrouping

In fact, the operational semantics for nested documents that do not perform restructuring is fairly simple: for each binding, construct a new element or attribute, and apply this rule recursively to all subelements/subbindings. However, this does not explain result patterns for computing, for example, the list of all title/author pairs or a bibliography that gives for each author all publication titles.

Therefore, we introduce rules for ungrouping and grouping bindings. In general, a binding for a pattern P is included as a subbinding within several enclosing patterns $P_1, \ldots, P_k$. Ungrouping is necessary if P is used in a result pattern which omits some P_i. Then, all the bindings for P_{i+1} are concatenated into one binding. If in addition to P more patterns $Q, R, \ldots$ are used, then first their cartesian product is computed, followed by concatenation. The use of the patterns $P, Q, R, \ldots$ in the result pattern then has the effect of projecting the corresponding column.

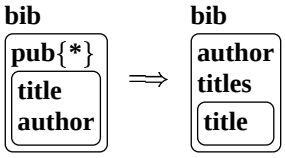
For example, assume the bibliographic entries are further grouped by nested elements **topic** and **year** (and, as before, by kind of publication, that is, **book**, ...). Consider now the following result patterns:



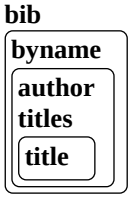
The first pattern keeps the original nesting structure and shows for each topic, year, and publication the list of authors. The second pattern shows the list of all authors for each topic and year; the author lists of all publications have been concatenated. In contrast, the third pattern concatenates all **pub**-bindings for different years and yields all publication entries grouped only by topic and yields for each publication the list of authors. The last pattern performs two independent groupings and yields the list of all authors for each year.

Although we can now perform "flattening" of data, it has still to be explained how to completely restructure data. For example, we might ask for a list of authors together with the titles of all their publications.

This can be achieved by the following rule:



The crucial point here is the omission of the **pub**-pattern (which performs ungrouping) together with the introduction of a new **titles**-elements (which causes grouping): after flattening the author/title information, we (at least conceptually) obtain an intermediate list of author/title tuples. Using only some of the columns of the intermediate relation (here, **author**) causes one such element being created for each different combination of values (here, for each author). The introduction of the nested **titles**-element performs grouping by creating a separate relation for the remaining fields. In this example, it is a relation of just one column, namely **title**, for each author. Thus, the result pattern creates an alternating sequence of **author**- and **titles**-elements, each of which contains a list of all **title**-elements for the preceding **author**-element.[2] To make the relationship between the titles and each author stronger and not dependent on the order of the elements in the result pattern, we can use an additional element for grouping them together, see the pattern to the right.



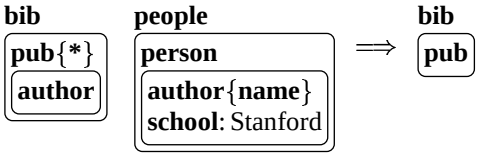
5 Some Advanced Topics

Due to space limitations, we were able to describe only the most basic concepts of Xing. However, we will give at least a brief sketch of some of the more advanced features in this section.

Joins. 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

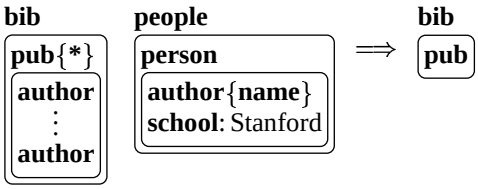
[2] If the **author**-element is swapped with the **titles**-element in the result pattern, the titles precede each author.

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 follows:

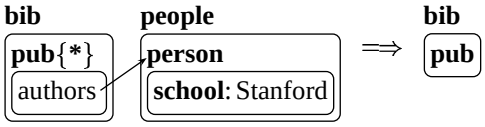


Note how the join-condition is expressed by an alias pattern: the **name**-element of *person* is renamed to **author**, and the use of equal variables signifies the condition that the bound values must be equal to become part of the result. If the names in the *person* document were, too, stored in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

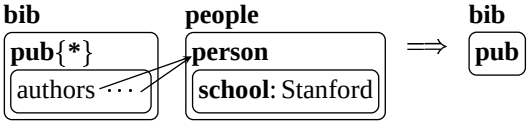
By default, the join on list elements like **author** is defined with an existential semantics, that is, in the above example, all publication are found that have at least one author from Stanford. This is equivalent to using an existential pattern on the **author**-variable. In contrast, to find publications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



Cross-Referencing. XML uses three special types of attributes to represent graph-like structures. Any element can have a single attribute of type ID that uniquely identifies that element. Other elements can refer to these identifiers through IDREF and IDREFS attributes, which establish links between the elements. Whereas an IDREF attribute always contains a single reference, an IDREFS attribute contains a sequence of references. Continuing the previous example, assume each person element in *person* is defined to have an ID attribute called "id", 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 element A references some other element B is then represented as an arrow $A \rightarrow B$. The above example could then expressed by:



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:



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 object 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.

Exploiting DTDs. DTDs define the attributes and the possible subelement structure of elements. It was a deliberate design decision that the query language capabilities do not depend on DTDs. On the other hand, if present, DTDs provide useful information about the data source that should be taken advantage of. There are at least two ways in which DTDs could be exploited:

First, in the user interface that offers means to build up documents by inserting tags and expanding them (that is, inserting boxes), DTDs can be very nicely utilized to offer all the possible attributes and/or subelements through pop-up menus. This offers on the one hand a convenient and fast way of constructing patterns, and on the other hand, it also provides automatic and incremental static type checking of patterns.

Second, DTDs can be used to optimize queries. For example, if a deep query asks for **author**-elements, a simple analysis of the DTD can reveal all those nodes in the document tree beyond which no **author**-elements can be found. This can be used to prune the search at exactly these points. Another example is to avoid unnecessary marking during matching with cross-references because a DTD can easily

tell which ID attributes are possibly referenced at all. There are certainly many more opportunities for optimization, and it is a very fruitful area of future research to apply and extend type-based optimization to XML and DTDs.

6 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

Concerning the implementation of the query back-end, we are currently investigating two possibilities: (1) tree matching, for example, based on the work in [15, 18] or (2) mapping to the relational model. The second option has the advantage that a huge body of work exists for query optimization and indexing, although it might not always be applicable (see [2] for trade-offs). On the other hand, the first alternative requires more implementation efforts, but might be in the end more efficient. Some results, for example, on indexing, are already available [20].

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] R. Baeza-Yates and G. Navarro. Integrating Contents and Structure in Text Retrieval. *ACM SIGMOD Record*, 25(1):67–79, 1996.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*, 1998. `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*. 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] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] H. Meuss. Indexed Tree Matching with Complete Answer Representations. In *4th Int. Workshop on Principles of Digital Document Processing*, LNCS 1481, pages 104–115, 1998.

[21] 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.

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

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

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

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

[26] 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.

[27] 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.