Querying XML documents with multi-dimensional markup

Peter Siniakov
siniakov@inf.fu-berlin.de
Database and Information Systems Group, Freie Universität Berlin
Takustr. 9, 14195 Berlin, Germany

Abstract

XML documents annotated by different NLP tools accommodate multi-dimensional markup in a single hierarchy. To query such documents one has to account for different possible nesting structures of the annotations and the original markup of a document. We propose an expressive pattern language with extended semantics of the sequence pattern, supporting negation, permutation and regular patterns that is especially appropriate for querying XML annotated documents with multi-dimensional markup. The concept of fuzzy matching allows matching of sequences that contain textual fragments and known XML elements independently of how concurrent annotations and original markup are merged. We extend the usual notion of sequence as a sequence of siblings allowing matching of sequence elements on the different levels of nesting and abstract so from the hierarchy of the XML document. Extended sequence semantics in combination with other language patterns allows more powerful and expressive queries than queries based on regular patterns.

1 Introduction

XML is widely used by NLP tools for annotating texts. Different NLP tools can produce overlapping annotations of text fragments. While a common way to cope with concurrent markup is using stand-off markup (Witt, 2004) with XPointer references to the annotated regions in the source document, another solution is to consolidate the annotations in a single document for easier processing. In that case concurrent markup has to be merged and accommodated in a single hierarchy. There are many ways to merge the overlapping markup so that different nesting structures are possible. Besides, the annotations have to be merged with the original markup of the document (e.g. in case of a HTML document). The problem of merging overlapping markup has been treated in (Siefkes, 2004) and we do not consider it here. Instead we focus on the problem of finding a universal querying mechanism for documents with multi-dimensional markup. The query language should abstract from the concrete merging algorithm for concurrent markup, that is to identify desired elements and sequences of elements independently from the concrete nesting structure.

The development of the query language was motivated by an application in text mining. In some text mining systems the linguistic patterns that comprise text and XML annotations (such as syntactic annotations, POS tags) made by linguistic tools are matched with semistructured texts to find desired information. These texts can be HTML documents that are enriched with linguistic information by NLP tools and therefore contain multi-dimensional markup. The linguistic annotations are specified by XML elements that contain the annotated text fragment as CDATA. Due to the deliberate structure of the HTML document the annotations can be nested in arbitrary depth and vice versa – the linguistic XML element can contain some HTML elements with nested text it refers to. To find a linguistic pattern we have to abstract from the concrete DTD and actual structure of the XML document ignoring irrelevant markup, which leads to some kind of "fuzzy" matching. Hence it is sufficient to specify a sequence

of text fragments and known XML elements (e.g. linguistic tags) without knowing by what elements they are nested. During the matching process the nesting markup will be omitted even if the sequence elements are on different nesting levels.

We propose an expressive pattern language with the extended semantics of the sequence pattern, permutation, negation and regular patterns that is especially appropriate for querying XML annotated documents. The language provides a rich tool set for specifying complex sequences of XML elements and textual fragments. We ignore some important aspects of a fully-fledged XML query language such as construction of result sets, aggregate functions or support of all XML Schema structures focusing instead on the semantics of the language.

Some modern XML query languages impose a relational view of data contained in the XML document aiming at retrieval of sets of elements with certain properties. While these approaches are adequate for database-like XML documents, they are less appropriate for documents in that XML is used rather for annotation than for representation of data. Taking the rather textual view of a XML document its querying can be regarded as finding patterns that comprise XML elements and textual content. One of the main differences when querying annotated texts is that the query typically captures parts of the document that go beyond the boundaries of a single element disrupting the XML tree structure while querying a database-like document returns its subtrees remaining within a scope of an element. Castagna (Castagna, 2005) distinguishes path expressions that rather correspond to the database view and regular expression patterns as complementary "extraction primitives" for XML data. Our approach enhances the concept of regular expression patterns making them mutually recursive and matching across the element boundaries.

2 Related Work

After publishing the XML 1.0 recommendation the early proposals for XML query languages focused primarily on the representation of hierarchical dependencies between elements and the expression of properties of a single element. Typically, hierarchical relations are defined along parent/child and ancestor/descendant axis as done in XQL and XPath. XQL (Robie, 1998) supports positional relations between the elements in a sibling list. Sequences of elements can be queried by "immediately precedes" and "precedes" operators restricted on the siblings. Negation, conjunction and disjunction are defined as filtering functions specifying an element. XPath 1.0 (Clark and DeRose, 1999) is closely related addressing primarily the structural properties of an XML document by path expressions. Similarly to XQL sequences are defined on sibling lists. Working Draft for Xpath 2.0 (Berglund et al., September 2005) provides support for more data types than its precursor, especially for sequence types defining set operations on them.

XML_QL (Deutsch et al., 1999) follows the relational paradigm for XML queries, introduces variable binding to multiple nodes and regular expressions describing element paths. The queries are resolved using an XML graph as the data model, which allows both ordered and unordered node representation. XQuery (Boag et al., 2003) shares with XML_QL the concept of variable bindings and the ability to define recursive functions. XQuery features more powerful iteration over elements by FLWR expression borrowed from *Quilt* (Chamberlin et al., 2001), string operations, "if else" case differentiation and aggregate functions. The demand for stronger support of querying annotated texts led to the integration of the full-text search in the language (Requirements, 2003) enabling full-text queries across the element boundaries.

Hosoya and Pierce propose integration of XML queries in a programming language (Hosoya and Pierce, 2001) based on regular patterns Kleene's closure and union with the "first-match" semantics. Pattern variables can be declared and bound to the corresponding XML nodes during the matching process. A static type inference system for pattern variables is incorporated in *XDuce* (Hosoya and Pierce, 2003) – a functional language for XML processing. *CDuce* (Benzaken et al., 2003) extends *XDuce* by an efficient matching al-

gorithm for regular patterns and first class functions. A query language *CQL* based on regular patterns of *CDuce* uses *CDuce* as a query processor and allows efficient processing of *XQuery* expressions (Benzaken et al., 2005). The concept of fuzzy matching has been inroduced in query languages for IR (Carmel et al., 2003) relaxing the notion of context of an XML fragment.

3 Querying by pattern matching

The general purpose of querying XML documents is to identify and process their fragments that satisfy certain criteria. We reduce the problem of querying XML to pattern matching. The patterns specify the query statement describing the desired properties of XML fragments while the matching fragments constitute the result of the query. Therefore the pattern language serves as the query language and its expressiveness is crucial for the capabilities of the queries. The scope for the query execution can be a collection of XML documents, a single document or analogously to XPath a subtree within a document with the current context node as its root. Since in the scope of the query there may be several XML fragments matching the pattern, multiple matches are treated according to the "all-match" policy, i.e. all matching fragments are included in the result set. The pattern language does not currently support construction of new XML elements (however, it can be extended adding corresponding syntactic constructs). The result of the query is therefore a set of sequences of XML nodes from the document. Single sequences represent the XML fragments that match the query pattern. If no XML fragments in the query scope match the pattern, an empty result set is returned.

In the following sections the semantics, main components and features of the pattern language are introduced and illustrated by examples. The complete EBNF specification of the language can be found on
`http://page.mi.fu-berlin.de/~siniakov/patlan`.

3.1 Extended sequence semantics

Query languages based on path expressions usually return sets (or sequences) of elements that are conform with the original hierarchical structure of the document. In not uniformly structured XML documents, though, the hierarchical structure of the queried documents is unknown. The elements we may want to retrieve or their sequences can be arbitrarily nested. When retrieving the specified elements the nesting elements can be omitted disrupting the original hierarchical structure. Thus a sequence of elements does no longer have to be restricted to the sibling level and may be extended to a sequence of elements following each other on different levels of XML tree.

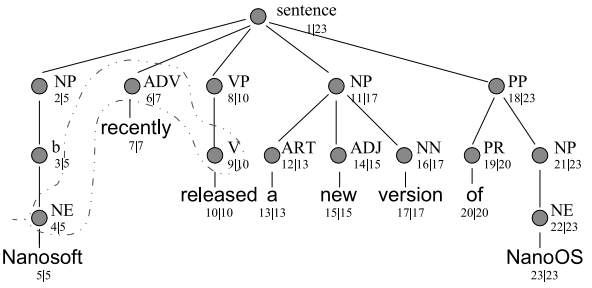

Figure 1: Selecting the sequence (NE ADV V) from a chunk-parsed POS-tagged sentence. XML nodes are labeled with preorder numbered $OID|right\ bound$ (maximum descendant OID)

To illustrate the semantics and features of the language we will use the mentioned text mining scenario. In this particular text mining task some information in HTML documents with textual data should be found. The documents contain linguistic annotations inserted by POS tagger and syntactic chunk parser as XML elements that include the annotated text fragment as a text node. The XML output of the NLP tools is merged with the HTML markup so that various nestings are possible. A common technique to identify the relevant information is to match linguistic patterns describing it with the documents. The fragments of the documents that match are likely to contain relevant information. Hence the problem is to identify the fragments that match our linguistic patterns, that is, to answer the query where the queried fragments are described by linguistic patterns. Linguistic patterns comprise sequences of text fragments and XML elements added by NLP tools and are specified in our pattern language. When looking for linguistic patterns in an annotated HTML docu-

ment, it cannot be predicted how the linguistic elements are nested because nesting depends on syntactic structure of a sentence, HTML layout and the way both markups are merged. Basically, the problem of unpredictable nesting occurs in any document with a heterogeneous structure. Let us assume we would search for a sequence of POS tags: `NE ADV V` in a subtree of a HTML document depicted in fig. 1. Some POS tags are chunked in noun (NP), verb (VP) or prepositional phrases (PP). Named entity "Nanosoft" is emphasized in boldface and therefore nested by the HTML element <b>. Due to the syntactic structure and the HTML markup the elements `NE, ADV` and `V` are on different nesting levels and not children of the same element. According to the extended sequence semantics we can ignore the nesting elements we are not interested in (NP_{OID2} and b_{OID3} when matching NE, VP_{OID8} when matching V) so that the sequence (NE_{OID4}, ADV_{OID6}, V_{OID9}) matches the sequence pattern `NE ADV V`, in short form `NE ADV V` $\cong$ (NE_4, ADV_6, V_9).

By the previous example we introduced the matching relation $\cong$ as a binary relation $\cong \subseteq P \times F$ where P is the set of patterns and F a set of XML fragments. An XML fragment f is a sequence of XML nodes $n_1 \dots n_n$ that belong to the subtree of the context node (i.e. the node whose subtree is queried, e.g. document root). Each XML node in the subtree is labeled by the pair $OID|right\ bound$. OID is obtained assigning natural numbers to the nodes during the preorder traversal. *Right bound* is the maximum OID of a descendant of the node – the OID of the rightmost leaf in the rightmost subtree. To match a sequence pattern an XML fragment has to fulfil four important requirements.

1. Consecutiveness: All elements of the sequence pattern have to match the consecutive parts of the XML fragment

2. Order maintenance: Its elements must be in the "tree order", i.e., the $OIDs$ of the nodes according to the preorder numbering schema must be in ascending order.

3. Absence of overlaps: No node in the sequence can be the predecessor of any other node in the sequence on the way to the root. E.g. `NP PP NP` $\ncong$ (NP_{11}, PP_{18}, NP_{21}) because PP_{18} is a predecessor of NP_{21} and therefore subsumes it in its subtree. The semantics of the sequence implies that a sequence element cannot be subsumed by the previous one but has to follow it in another subtree. To determine whether a node m is a predecessor of the node n the $OIDs$ of the nodes are compared. The predecessor must have a smaller OID according to the preorder numbering scheme, however any node in left subtrees of n has a smaller OID too. Therefore the *right bounds* of the nodes can be compared since the *right bound* of a predecessor will be greater or equal to the *right bound* of n while the *right bound* of any element in the left subtree will be smaller:

$$pred(m,n) = \\ OID(m)<OID(n) \wedge rightBound(m) \geq rightBound(n)$$

4. Completeness: XML fragment must not contain any gaps, i.e. there should not be a node that is not in the XML fragment, not predecessor of one of the nodes, whose OID however lies between the $OIDs$ of the fragment nodes. Since such a node is not a predecessor, it must be an element of the sequence; otherwise it is omitted and the sequence is not complete. Hence, the pattern `V NP NP` $\ncong$ (V_9, NP_{11}, NP_{21}) because the node PR_{19} lying between NP_{11} and NP_{21} is not a predecessor of any of the fragment nodes and not an element of the fragment. If the nodes lying between NP_{11} and NP_{21} cannot be exactly specified, we can use wildcard pattern (see sec. 3.3) to enable matching: `V NP * NP` $\cong$ (V_9, NP_{11}, PR_{19}, NP_{21}):

Using these requirements we can formally specify the semantics of the sequence:
Let $s = s_1 \dots s_k$ be a sequence pattern and $f = n_1 \dots n_n$ the matching XML fragment.

$$s \cong f \Leftrightarrow$$

$$(I) \quad s_1 \cong (n_1 \dots n_i),\ s_2 \cong (n_{i+1} \dots n_j), \dots, s_k \cong (n_l \dots n_n)$$

$$(II) \quad \forall\ 1 \leq i < n\ OID(n_i) < OID(n_{i+1})$$

$$(III) \quad \nexists\ 1 \leq i < n\ pred(n_i, n_{i+1})$$

$$(IV) \quad \forall\ 1 \leq i < n\ \nexists\ m\ OID(n_i) < OID(m) < OID(n_{i+1}) \wedge \\ \wedge \neg pred(m, n_{i+1})$$

The fourth requirement stresses the important aspect of "exhaustive" sequence: we are interested in a certain sequence of known elements that can be arbitrarily nested and captured by some elements that are irrelevant for our sequence (e.g. html layout elements when searching for a sequence of linguistic elements). We call such a sequence an *exhaustive non-sibling sequence* (*ENSS*). It is exhaustive because all predecessors omitted during the matching are covered at some level by the matching descendants so that there is no path to a leaf of the predecessor subtree that leads through an unmatched node. If such a path existed, the fourth requirement would not be met. If the sequence does not begin at the leftmost branch or does not end at the rightmost branch of an omitted predecessor, the subtree of the respective predecessor is not fully covered. In `ADJ NN PR` $\cong$ (ADJ_{14}, NN_{16}, PR_{19}) the omitted predecessors NP_{11} and PP_{18} are not completely a part of the sequence because they have descendants outside the sequence borders. Nevertheless the sequence is exhaustive since there is no path to a leaf through an unmatched node within its borders.

Another important aspect of *ENSS* is that it can match XML fragments across the element borders. XPath imposes a query context by specifying the path expression that usually addresses a certain element, XQuery restricts it indirect by iterating over and binding variables to certain nodes. Matching *ENSS* there is no additional restriction of the query scope, that is, the sequence can begin and end at any node provided that the *ENSS* requirements are met. The dashed line in the fig. 1 points up the region covered by the sample sequence.

According to the specification of the sequence pattern in the pattern language (cf. appendix **??**):
Pattern ::= *Pattern*$'$ $'$* *Pattern*
any pattern can be the element of the sequence. Therefore the sequence can also contain textual elements, which is especially important when processing annotated texts. Textual nodes represent leaves in an XML tree and are treated as other XML nodes so that arbitrary combinations of XML elements and text are possible: `"released" NP "of" NE` $\cong$ ("released"$_{10}$, NP_{11}, "of"$_{20}$, NE_{22})

Exhaustive sequence allows a much greater abstraction from the DTD of a document than the usually used sequence of siblings. The expressiveness of the language significantly benefits from the combination of backtracking patterns (cf. sec. 3.3) with exhaustive sequence.

3.2 Specification of XML nodes

Patterns matching single XML nodes are the primitives that the more complex patterns are composed from. The pattern language supports matching for document, element, attribute, text and CDATA nodes while some DOM node types such as entities and processing instructions are not supported. Some basic patterns matching element and text nodes have been already used as sequence elements in the previous section. Besides the simple addressing of an element by its name it is possible to specify the structure of its subtree:
Pattern ::=$'$ \$'$XML-Tag($'$[$'$Pattern$'$]$'$)?
A pattern specifying an element node will match if the element has the name corresponding to the XML-Tag and the pattern in the square brackets matches the XML fragment containing the sequence of its children. E.g. `\PP[ PR NE]` $\cong$ (PP_{18}) because the name of the element is identical and `PR NE` $\cong$ (PR_{19}, NE_{22}). As this example shows, the extended sequence semantics applies also when the sequence is used as the inner pattern of another pattern. Therefore the specification of elements can benefit from the *ENSS* because we again do not have to know the exact structure of their subtrees, e.g. their children, but can specify the nodes we expect to occur in a certain order.

Attribute nodes can be accessed by element pattern specifying the attribute values as a constraint: `\V {@normal="release"}` $\cong$ (V_9), assumed that the element V_9 has the attribute "normal" that stores the principal form of its textual content. Besides equality tests, numeric comparisons and boolean functions on string attribute values can be used as constraints.

Patterns specifying textual nodes comprise quoted strings:
Pattern ::= QuotedString
and match a textual node of an XML element if it has the same textual content as the quoted string. Textual patterns can be used as ele-

ments of any other patterns as already demonstrated in the previous section. An element may be, for instance, described by a complex sequence of text nodes combined with other patterns: `\sentence[NE * \V{@normal=release} \NP[* "new" "version"] "of" NE *]` $\cong$ (sentence$_1$) The pattern above can already be used as a linguistic pattern identifying the release of a new product version.

3.3 Backtracking patterns and variables

In contrast to the database-like XML documents featuring very rigid and repetitive structures annotated texts are distinguished by a very big structural variety. To handle this variety one needs patterns that can cover several different cases "at once". So called backtracking patterns have this property and constitute therefore a substantial part of the pattern language. Their name comes from the fact that during the matching process backtracking is necessary to find a match.

The pattern language features complex and primitive patterns. Complex patterns consist of at least one inner element that is a pattern itself. Primitive patterns are textual patterns or XML attribute and element specifications if the specification of the inner structure of the element is omitted, e.g. `"released", NP`. If at least one of the inner patterns does not match, the matching of the complex pattern fails. Backtracking patterns except for wildcard pattern are complex patterns.

Let us assume, we look for a sequence `"released" NE` and do not care what is between the two sequence elements. In the subtree depicted in fig. 1 no XML fragment will match because there are several nodes between "released"$_{10}$ and NE$_{22}$ and the completeness requirement is not met. If we include the wildcard pattern in the sequence, `"released" * NE` $\cong$ ("released"$_{10}$ NP$_{11}$ PR$_{19}$ NE$_{22}$), the wildcard pattern matches the nodes lying between V$_9$ and NE$_{22}$. Thus, every time we do not know what nodes can occur in a sequence or we are not interested in the nodes in some parts of the sequence, we can use wildcard pattern to specify the sequence without losing its completeness. Wildcard pattern matches parts of the sequence that are in turn sequences themselves. Therefore it matches only those XML fragments that fulfil the *ENSS* requirements II-IV. Since there are often multiple possibilities to match a sequence on different levels, wildcard matches nodes that are at the highest possible level such as NP$_{11}$ in the previous example.

If one does not know whether an XML fragment occurs, but wants to account for both cases the option pattern should be used:
Pattern ::=′ (′Pattern′)?′
Pattern ::=′ (′Pattern′)*′
Kleene closure differs from the option by the infinite number of repetitions. It matches a sequence of any number of times repeated XML fragments that match the inner pattern of the Kleene closure pattern. Since Kleene closure matches sequences, the *ENSS* requirements have to be met by matching XML fragments. Let $O = (\mathsf{p})?$ be an option, $K = (\mathsf{p})^*$ a Kleene closure pattern, $f \in F$ an XML fragment:

$$O \cong f \quad \Leftrightarrow \quad \mathsf{p} \cong f \ \vee \ \{\} \cong f$$
$$K \cong f \quad \Leftrightarrow \quad \{\} \cong f \ \vee \ \mathsf{p} \cong f \ \vee \ \mathsf{p}\ \mathsf{p} \cong f \ \vee \ \ldots$$

where f fulfils *ENSS* requirements I-IV.
The option pattern matches either an empty XML fragment or its inner pattern.

An alternative occurrence of two XML fragments is covered by the union pattern:
Pattern ::=′ (′Pattern(′|′Pattern)+′)′
Different order of nodes in the sequence can be captured in the permutation pattern:
Pattern ::=′ (′Pattern Pattern+′)%′
Let $U = (\mathsf{p}_1|\mathsf{p}_2)$ be a union pattern, $P = (\mathsf{p}_1, \ldots, \mathsf{p}_n)\%$ a permutation pattern

$$U \cong f \quad \Leftrightarrow \quad \mathsf{p}_1 \cong f \vee \mathsf{p}_2 \cong f$$
$$P \cong f \quad \Leftrightarrow \quad \mathsf{p}_1\ \mathsf{p}_2 \ldots \mathsf{p}_n \cong f \vee \mathsf{p}_1\ \mathsf{p}_2 \ldots \mathsf{p}_n\ \mathsf{p}_{n-1} \cong f \vee \cdots$$
$$\cdots \vee \mathsf{p}_1\ \mathsf{p}_n \ldots \mathsf{p}_2 \cong f \vee \cdots \vee \mathsf{p}_n\ \mathsf{p}_{n-1} \ldots \mathsf{p}_2\ \mathsf{p}_1 \cong f$$

Permutation can not be expressed by regular constructs and is therefore not a regular expression itself.

The backtracking patterns can be arbitrarily combined to match complex XML fragments. E.g. the pattern `((PP | PR)? NP)%` matches three XML fragments: (NP$_2$), (NP$_{11}$, PP$_{18}$) and (PR$_{19}$, NP$_{21}$). Using the backtracking patterns recursively enlarges the expressivity of the patterns a lot allowing to specify very complex and variable structures without significant syntactic effort.

Variables can be assigned to any pattern
Pattern ::= Pattern′ =:′ String
accomplishing two functions. Whenever a variable is referenced within a pattern by the reference pattern Pattern ::=′ $′String′$′ it evaluates to the pattern it was assigned to. The pattern `(NP)*=:noun_phrase * $noun_phrase$` ≅ (NP_2, ADV_6, VP_8, NP_{11}) so that the referenced pattern matches NP_{11}. A pattern referencing the variable v matches XML fragments that match the pattern that has been assigned to v. To make the matching results more persistent and enable further processing variables can be bound to the XML fragment that matched the pattern the variable is assigned to. After matching the pattern `\sentence[NE=:company * \V{@normal=release} \NP[* "new" "version"] "of" NE=:product *]` ≅ (sentence_1) the variable `company` refers to NE_4(Nanosoft) and `product` is bound to NE_{22}(NanoOS). The relevant parts of XML fragment can be accessed by variables after a match has been found. Assigning variable to the wildcard pattern can be used to extract a subsequence between two known nodes: `"released" * =:direct_object "of"` ≅ ("released"$_{10}$ NP_{11} "of"$_{20}$) with the variable `direct_object` bound to NP_{11}.
Let $A = \mathsf{p} =: \mathsf{v}$ be an assignment pattern:

$$A \cong f \Leftrightarrow \mathsf{p} \cong f$$

Matching backtracking patterns can involve multiple matching variants of the same XML fragment, which usually leads to different variable bindings for each matching variant. As opposed to multiple matchings when different fragments match the same pattern discussed above, the first-match policy is applied when the pattern ambiguously matches a XML fragment. For instance,two different matching variants are possible for the pattern `(NP)?:=noun_phrase (NP |PR)*:=noun_prep` ≅ (NP_{11}, PR_{19}). In the first case `(NP)?:=noun_phrase` ≅ (NP_{11}) so that `noun_phrase` is bound to NP_{11} and `noun_prep` to PR_{19}. In the second case `(NP)?:=noun_phrase` ≅ {} and `(NP | PR)*:=noun_prep` ≅ (NP_{11}, PR_{19}) so that `noun_phrase` is bound to {} and `noun_prep` to (NP_{11}, PR_{19}). In such cases the first found match is returned as the final result. The order of processing of single patterns is determined by a convention.

3.4 Negation

When querying an XML document it is often useful not only to specify what is expected but also to specify what should not occur. This is an efficient way to exclude some unwanted XML fragments from the query result because sometimes it is easier to characterize an XML fragment by not wanted rather than desirable properties. Regular languages (according to Chomsky's classification) are not capable of representing that something should not appear stating only what may or has to appear. In the pattern language the absence of some XML fragment can be specified by negation .

As opposed to most XML query languages negation is a pattern and not a unary boolean operator. Therefore it has no boolean value, but matches the empty XML fragment. Since the negation pattern specifies what should not occur, it does not "consume" any XML nodes during the matching process so that we call it "non-substantial" negation. The negation pattern `!(p)` matches the empty XML fragment if its inner pattern `p` does not occur in the current context node. To underline the difference to logical negation, consider the double negation. The double negation `!(!(p))` is not equivalent to p, but matches an empty XML element if `!(p)` matches the current context node, which is only true if the current context node is empty. Since the negation pattern only specifies what should not occur, the standalone usage of negation is not reasonable. It should be used as an inner pattern of other complex patterns. Specifying a sequence `VP *=:wildcard_1 !(PR) *=:wildcard_2 NP` we want to identify sequences starting with VP and ending with NP where PR is not within a sequence. Trying to find a match for the sequence starting in VP_8 and ending in NP_{21} there are multiple matching variants for wildcard patterns. Some of them enable the matching of the negation pattern binding PR to one of the wildcards, e.g. `wildcard_1` is bound to (NP_{11}, PR_{19}), `!(PR)` ≅ {}, `wildcard_2` is bound to {}. However, there

is a matching variant when the negated pattern is matched with PR_{19} (`wildcard_1` is bound to NP_{11}, `wildcard_2` is bound to {}). We would certainly not want the sequence (VP_8, NP_{11}, PR_{19}, NP_{21}) to match our pattern because the occurrence of PR in the sequence should be avoided. Therefore we define the semantics of the negation so that there is no matching variant that enables the occurrence of negated pattern:
Let P_1 !(p) P_2 be a complex pattern comprising negation as inner pattern. P_1 and P_2 are the left and right syntactic parts of the pattern and may be not valid patterns themselves (e.g. because of unmatched parentheses). The pattern obtained from the concatenation of both parts P_1 P_2 is a valid pattern because it is equivalent to the replacing of the negation by an empty pattern.

$$\mathsf{P}_1\ !(\mathsf{p})\ \mathsf{P}_2 \cong f \Leftrightarrow$$
$$\mathsf{P}_1\ \mathsf{p}\ \mathsf{P}_2 \ncong f \wedge \mathsf{P}_1\ \mathsf{P}_2 \cong f$$

Requiring $\mathsf{P}_1\ \mathsf{p}\ \mathsf{P}_2 \ncong f$ guarantees that no matching variant exists in that the negated pattern p occurs. Since !(p) matches an empty fragment, the pattern $\mathsf{P}_1\mathsf{P}_2$ has to match complete f. It is noteworthy that the negation is the only pattern that influences the semantics of a complex pattern as its inner pattern. Independent of its complexity any pattern can be negated allowing very fine-grained specification of undesirable XML fragments.

4 Conclusion

XML documents with multi-dimensional markup feature a heterogeneous structure that depends on the algorithm for merging of concurrent markup. We present a pattern language that allows to abstract from the concrete structure of a document and formulate powerful queries. The extended sequence semantics allows matching of sequences across element borders and on different levels of the XML tree ignoring nesting levels irrelevant for the query. The formal specification of the sequence semantics guarantees that the properties of "classic" sibling sequence such as ordering, absence of gaps and overlaps between the neighbors are maintained. The combination of fully recursive backtracking patterns with the *ENSS* semantics allows complex queries reflecting the complicated positional and hierarchical dependencies of XML nodes within a multi-dimensional markup. Negation enhances the expressivity of the queries specifying an absence of a pattern in a certain context.

References

V. Benzaken, G. Castagna, and A. Frisch. 2003. CDuce: an XML-centric general-purpose language. In *In ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63.

V. Benzaken, G. Castagna, and C. Miachon. 2005. A full pattern-based paradigm for XML query processing. In *Proceedings of the 7th Int. Symposium on Practical Aspects of Decl. Languages*, number 3350.

A. Berglund, S. Boag, D. Chamberlin, M. Fernndez, M. Kay, J. Robie, and J. Simon. September 2005. XML Path Language (XPath) 2.0. `http://www.w3.org/TR/2005/WD-xpath20-20050915/`.

S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. 2003. XQuery 1.0: An XML Query Language. `http://www.w3c.org/TR/xquery`.

David Carmel, Yoelle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. 2003. Searching XML documents via XML fragments. In *SIGIR '03: Proceedings of the 26th annual int. ACM SIGIR conference*, pages 151–158, New York,USA. ACM Press.

G. Castagna. 2005. Patterns and types for querying XML. In *DBPL - XSYM 2005 joint keynote talk*.

Don Chamberlin, Jonathan Robie, and Daniela Florescu. 2001. Quilt: An XML query language for heterogeneous data sources. *LNCS*, 1997:1–11.

J. Clark and S. DeRose. 1999. XML Path Language (XPath). `http://www.w3.org/TR/Xpath`.

Alin Deutsch, Mary F. Fernandez, D. Florescu, A.Y. Levy, and D. Suciu. 1999. A query language for XML. *Computer Networks*, 31(11-16):1155–1169.

H. Hosoya and P.C. Pierce. 2001. Regular expression patern matching for XML. In *In POPL '01, 25th Symposium on Principles of Prog. Languages*.

Haruo Hosoya and Benjamin C. Pierce. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148.

XQuery and XPath Full-Text Requirements. 2003. `http://www.w3.org/TR/2003/WD-xquery-full-text-requirements-20030502/`.

Jonathan Robie. 1998. The design of XQL. `http://www.ibiblio.org/xql/xql-design.html`.

C. Siefkes. 2004. A shallow algorithm for correcting nesting errors and other well-formedness violations in XML-like input. In *Extreme Markup Languages*.

A. Witt. 2004. Multiple hierarchies: new aspects of an old solution. In *Extreme Markup Languages 2004*.