6 Pattern Unification by Querying XML: A Pattern Based XML Query Language

Our approach to IE is based on learning patterns recurring in natural language for expression of certain information. The learned patterns are not supposed to capture exact phrases and expressions, but to be able to include essential parts and features of expressions allowing to abstract from irrelevant parts. Patterns consist not only in lexical elements and their positional dependencies, but include also linguistic and structural features obtained during the preprocessing. Natural language itself is not suitable for specification of patterns since it does not offer any formal means for abstraction and capturing non-lexical information. The pattern specification language has therefore to fulfil following requirements:

- ▷ **Formal Specification**. A formal base of the pattern specification language facilitates the algorithmic induction of extraction patterns. Furthermore, it allows to formally define the generalizing operations.
- ▷ **Expressivity**. Some rule-based approaches restrict the range of patterns to a subset of regular expressions and are forced to simplify the modeling of natural language taking in account that non-trivial language structures cannot be expressed. The pattern specification language has to feature expressive language elements to adequately reflect the variety of natural language.
- ▷ **Support for linguistic features and XML**. Since patterns incorporate linguistic and structural information, pattern language should feature components for expression of structural and linguistic properties and allow to interleave and nest different elements. Since the preprocessed documents are in XML format, the language has also to support XML syntax.

To comply with the requirements we defined a context-free pattern specification language (refer to appendix A) extending its expressivity by non-regular negation and permutation patterns and including XML support. The pattern language has a variety of features for handling syntactic and lexical diversity of a natural language. Particularly, it can be specified by the negation operator that certain lexical or syntactic sequence must not occur in the text. Various possibilities of information expression in the natural language like repetition, optional occurrence or occurrence of lexical or syntactic entities in the arbitrary order are covered by respective operators as well. Moreover, there are patterns comprising sets of synonyms or matching all inflections of a word by specifying the principal form. To enable the identification of relevant information a pattern encoding a relevant text fragment that should be extracted can be referenced by a variable (see examples in sec. 4.1). Matching text fragments are assigned to the variable and can be accessed for extraction.[1] Patterns referenced by a variable can also be reused in other patterns. In chapters 7 and 8 we will explain how the linguistic patterns are captured in the pattern language.

The pattern language significantly distinguishes our approach from other rule-based approaches to IE. Whisk [Sod99] and LP^2 [Cir01a] use a much more restrictive notion of a pattern: while Whisk employs only a subset of regular expressions encoding certain syntactic groups, LP^2 regards a fix context window with a seven predefined features independent of the syntactic and semantic complexity of texts and extracted fragments. Rapier [Cal98] does not restrict the length of the context window, uses though only three features (POS tag, lexical form and semantic class) to describe each token. Setting no limits for the number of features, providing XML support and a very rich set of general patterns and allowing their full recursivity we establish a more powerful and flexible pattern model for natural language texts. Facilitating the actual extraction by the unification of variables we introduce a novel extraction technique allowing the immediate access to the extracted contents after the unification has been completed.

So far we have established a view of the pattern language derived from the IE task: patterns serve for modeling the natural language in order to encode typical expression forms. On the other hand, patterns are used for localizing the desired information in the document by matching them with the text. Since the preprocessed texts are XML documents, the matching between the patterns and XML structure has to be established. This opens a totally different view of patterns as XML queries that retrieve XML fragments whose structure, textual content and linguistic elements correspond with the specification in the pattern. Due to the dualistic nature of the patterns the pattern specification language can be regarded as an independent XML query language. The next section describes the pattern language and explains its semantics from the perspective of XML querying.

6.1 Pattern Specification Language in the Role of XML Query Language

As we have already mentioned, extraction patterns include not only textual elements, but also linguistic and structural features and sometimes original markup of the document. Patterns cannot be looked for in the original text documents because they do not contain the information obtained during preprocessing. Therefore preprocessed documents serve as the basis for pattern unification.

During the preprocessing different NLP tools can produce overlapping annotations of text fragments. While a common way to cope with concurrent annotations is using stand-off markup [Wit04] with XPointer references to the annotated regions in the source document, pattern matching requires a consolidation of an-

[1] Since the eventual goal of pattern matching is to establish variable bindings with fragments that should be extracted, we borrowed the concept of variable “unification” from the logic so that the process of pattern matching coupled with the binding of pattern variables is referred to as “**pattern unification**”.

notations in a single document. This means that concurrent markup has to be merged and accommodated in a single hierarchy during the preprocessing. 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 at the preprocessing stage has been treated in [Sie04] and we do not consider it here. Instead we focus on the problem of finding a universal pattern matching mechanism for documents with multi-dimensional markup. Looking on the problem from the XML point of view we need a query language that is able to 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.

Due to the arbitrary structure of the HTML documents the annotations can be nested in arbitrary depth and vice versa – the linguistic XML elements 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 [Cas05] 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.

6.2 Existing XML Query Languages

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 [Rob98] 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 [Cla99] 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 [Ber05] provides support for more data types than its precursor, especially for sequence types defining set operations on them.

Bird et al. recognize the deficiencies of XPath in processing of XML annotated text documents [Bir05] and extend it to support typical queries on linguistic annotations. The proposed language LPath can express important horizontal positional relations within the sequence and also between the different levels of linguistic elements.

A recently released tree query language for syntactically annotated texts Tregex [Lev06] enables queries based on two basic relations dominance and precedence. It includes also transitive dependencies featuring a Kleene's closure, which however can include only one node. Boolean operations can be applied to the relations, but not to the tree nodes itself. The "horizontal" navigation is established by immediate and transitive precedence over a closure of nodes, each of which has to match the same pattern. In contrast to LPath this model is too weak to express complex sequences dispersed by arbitrary nesting structures.

XML_QL [Deu99] 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 [Boa03] 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* [Cha01], 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 [Req03] enabling full-text queries across the element boundaries.

Hosoya and Pierce propose integration of XML queries in a programming language [Hos01] 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* [Hos03] – a functional language for XML processing. *CDuce* [Ben03] extends *XDuce* by an efficient matching algorithm 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 [Ben05]. The concept of fuzzy matching has been introduced in query languages for IR [Car03] relaxing the notion of context of an XML fragment.

6.3 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 of pattern unification with XML documents resulting from the preprocessing described in the previous chapters. 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 document, 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.

The complete EBNF specification of the language can be found in appendix A.

6.3.1 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. A similar relation between two XML elements called "immediately follows" ("immediately precedes") has been introduced by Bird et al. [Bir05] to overcome the hierarchical barrier for expression of positional dependencies of elements on different linguistic layers (e.g. noun phrase and a part of speech).

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. 6.1. 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

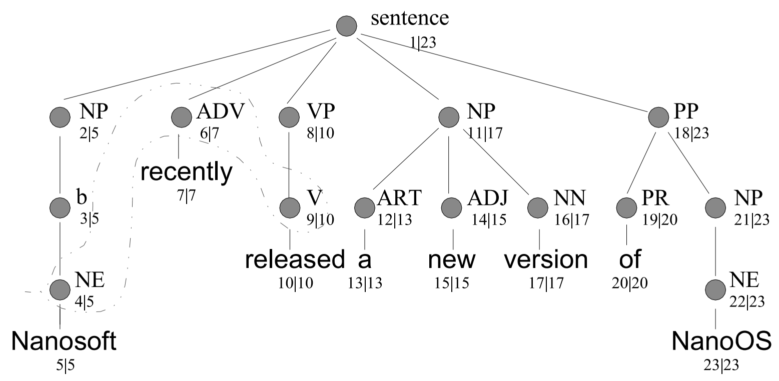

Figure 6.1: Selecting the sequence (NE ADV V) from a chunk-parsed POS-tagged sentence. XML nodes are labeled with preorder numbered $OID|right\ bound$ (maximum descendant OID)

to 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 $n_1 \ldots n_n$ 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. 6.3.3) 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 $s = s_1 \ldots s_k$ be a sequence pattern and $f = n_1 \ldots n_n$ the matching XML fragment.

$$s \cong f \Leftrightarrow$$

$$\begin{array}{ll}
(I) & s_1 \cong (n_1 \ldots n_i),\ s_2 \cong (n_{i+1} \ldots n_j), \ldots, s_k \cong (n_l \ldots n_n) \\
(II) & \forall\, 1 \leq i < n\ OID(n_i) < OID(n_{i+1}) \\
(III) & \nexists\, 1 \leq i < n\ pred(n_i, n_{i+1}) \\
(IV) & \forall\, 1 \leq i < n\ \ \nexists\, m\ OID(n_i) < OID(m) < OID(n_{i+1}) \wedge \neg pred(m, n_{i+1})
\end{array}$$

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. 6.1 points up the region covered by the sample sequence.

According to the specification of the sequence pattern in the pattern language (cf. appendix A):

$$\mathsf{Pattern} ::= \mathsf{Pattern}'\ '^{*}\ \mathsf{Pattern}$$

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. 6.3.3) with exhaustive sequence.

6.3.2 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 elements 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.

6.3.3 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. 6.1 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:

$\mathsf{Pattern} ::='\ ('\mathsf{Pattern}')?'$
$\mathsf{Pattern} ::='\ ('\mathsf{Pattern}')^{*\prime}$

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. As opposed to *Tregex* [Lev06] Kleene closure pattern builds a transitive closure over XML fragments with arbitrary complexity and not just a single node.
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 fulfills *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. Different order of nodes in the sequence can be captured in the permutation pattern:

$\mathsf{Pattern} ::='\ ('\mathsf{Pattern}('|'\mathsf{Pattern})+')'$
$\mathsf{Pattern} ::='\ ('\mathsf{Pattern\ Pattern}+')\%'$

Let $U = (\mathsf{p}_1|\mathsf{p}_2)$ be a union pattern, $P = (\mathsf{p}_1, \ldots, \mathsf{p}_n)\%$ a permutation pattern

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

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

$\mathsf{Pattern} ::= \mathsf{Pattern}'=:'\ \mathsf{String}$

accomplishing two functions. Whenever a variable is referenced within a pattern by the reference pattern

$\mathsf{Pattern} ::='\ \$'\mathsf{String}'\$',$

it evaluates to the pattern it was assigned to. The pattern `(NP)*=:noun_phrase * $noun_phrase$` $\cong$ (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 *]` $\cong$ (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"` $\cong$ ("released"$_{10}$ NP$_{11}$ "of"$_{20}$) with the variable `direct_object` bound to NP$_{11}$.
Let $A = \mathtt{p} =: \mathtt{v}$ be an assignment pattern:

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

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` $\cong$ (NP$_{11}$, PR$_{19}$). In the first case `(NP)?:=noun_phrase` $\cong$ (NP$_{11}$) so that `noun_phrase` is bound to NP$_{11}$ and `noun_prep` to PR$_{19}$. In the second case `(NP)?:=noun_phrase` $\cong$ $\{\}$ and `(NP | PR)*:=noun_prep` $\cong$ (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 backtracking branches is therefore crucial for appropriate variable bindings and hence for correct identification of information. The order of processing should therefore enable the most reliable and reasonable first match. In case of the wildcard, for example, the backtracking begins trying to match an empty fragment increasing it step by step allowing so to match the following patterns that generally specify the context of or extracted information itself more precisely, e.g. in
`* (V)?=:action` $\cong$ (NP$_2$, ADV$_6$, V$_9$) the variable `action` is bound to V$_9$.

6.3.4 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 (e.g. *Tregex*) 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)` $\cong$ {}, `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.

6.4 Unification Algorithm

In the previous sections we introduced the formal semantics of the pattern language, but did not explain how the patterns can be efficiently unified with XML documents. This section presents a very fast stack-based algorithm for pattern matching and unification with XML documents.

Besides the expressivity one of the main criteria for the goodness of a query language is the efficiency of the query processing. Regular patterns of XDuce and CDuce can be efficiently processed by FSMs. However, these languages loose the expressive power lacking such important patterns as wildcard and negation. Negation alone extends a pattern language beyond the expressive power of regular languages so that FSMs cannot be used as the computational model for pattern matching.

We propose a pattern matching algorithm that leverages the concept of a stack for processing of both patterns and XML documents and features an efficient backtracking mechanism. To assure the correct order of processing the stack model adequately reflects the recursive structure of patterns and hierarchical structure of XML documents. Patterns are processed on the pattern stack, XML documents – on the element stack. Once a pattern has been matched, it is removed from the stack. If a complex pattern is processed, its inner patterns are put on the stack according to the semantics of the pattern. Both stacks capture the current state of the matching process storing parts of a pattern or a XML fragment that are not yet matched.

To handle the extended sequence semantics the system tries to match the higher nodes of the hierarchy and if it fails, descends to the next lower level putting the children of the currently processed element in reversed order on the stack. This process continues iteratively until a match is achieved or a leaf of the XML tree is reached. Once the descendants of a node n are put on the stack, the matching process continues on the level of the respective descendant and can only return to the level of n after all its descendants are matched. This guarantees that the *ENSS* requirements III-IV are fulfilled.

6.4.1 Unification of Negation Pattern

Since the negation changes the matching semantics of the "ancestor" and inner patterns that are processed before and after the negation pattern, its handling is the most challenging part of the algorithm. Let $N = \mathsf{P_1\ !(p)\ P_2}$ be a negation pattern. On the one hand, if the pattern $\mathsf{P_1\ p\ P_2}$ matches at some point of backtracking, the N will not match and no further backtracking in the ancestor patterns is necessary. On the other hand, if at some point during the backtracking of ancestor patterns $\mathsf{P_1\ p\ P_2}$ does not match, the backtracking will have to continue in order to exclude the possibility that $\mathsf{P_1\ p\ P_2}$ matches. To control the backtracking behavior of ancestor patterns a special returned value is used. *Matching result* is a tuple (*success*, *backtracking*) that consists of two boolean values. *success* denotes whether a match has been found and *backtracking* – whether the backtracking should be continued. Depending on the results of matching $\mathsf{P_1\ p\ P_2}$ the appropriate matching result is returned. If $\mathsf{P_1\ p\ P_2}$ matched, $(false,\ false)$ is returned indicating that the pattern N did not match and no further backtracking is necessary, otherwise (*true*, *true*) is returned indicating that in this special backtracking branch N matches, but the backtracking should continue.

6.4.2 Handling Backtracking and Assignment Patterns

In case that the backtracking patterns are processed, copies of the both stacks have to be made before the backtracking is started. If a match is not found in one of backtracking branches, the stacks have to be reset to the state before the backtracking began. Furthermore due to continuing backtracking after a match in case of negation the first match and the stacks have to be backed up to restore them after the backtracking ends.

Assignment patterns require a special treatment of their inner patterns. After the inner pattern of the assignment pattern has been put on the stack there is no possibility to determine where it ends. Since the variable is bound to the XML fragment that matches its assigned pattern, it is important to know the borders of the inner pattern. For this purpose a special *pattern end marker* is put on the pattern stack. Every time the pattern end marker is removed, the matching XML fragment is bound to the variable. To store the variable bindings an environment is established. Variable bindings are twofold storing the assigned pattern and matching XML fragments. If there are multiple matches of the pattern in the document, the environment stores all matching XML fragments in a list of variable bindings. When the reference pattern is processed, the value of the variable storing the assigned pattern is taken from the environment and put on the pattern stack.

In the appendix B we present the unification algorithm for sequence, negation and selected backtracking patterns in pseudocode. For the sake of simplicity we omitted important aspects of the language such as variable bindings, multiple matches etc.

6.4.3 Assessment of Time Complexity

Time Complexity of Matching Non-backtracking Patterns and their Sequences

The time complexity of the algorithm depends to a high degree on the number of nodes in the subtree of the context node and the number and kind of backtracking patterns. For any non-backtracking patterns the match is established traversing the subtree of the context node until the matching node is found. This operation requires constant time in the best case and $\frac{n}{2}$ in the average case, where n is the number of randomly distributed nodes in the subtree of the context node. However, linguistically preprocessed documents feature high locality of syntactic XML elements, so that the distribution of nodes is not random. In this case non-backtracking pattern (e.g. XML element pattern) will match in one of the first branches so that the complexity is proportional to the height of the subtree of the context node corresponding in the average case to $log\ n$.

The time complexity for the sequence of non-backtracking patterns lies in the dimension of $O(n)$. Assuming that the match of the first sequence element consumed m_1 first nodes in the subtree of the context node the problem size for all subsequent elements is reduced to $O(n - m_1)$ since according to the ENSS "consumed" nodes cannot participate in the matching of subsequent elements. Matching the sequence of k elements terminates when the last sequence element is matched against the node within the context subtree, i.e. $\sum_{i=1}^{k} m_i \leq n$ or if one of the sequence elements could not be matched. If j-th sequence element could not be matched, the total time complexity still yields $\sum_{i=1}^{j-1} m_i + m'_j \leq n$, where m'_j is the number of consumed nodes during the unsuccessful attempt to match j-th element. Therefore even in the worst case the subtree of the context node can only be traversed once during the matching of the sequence, which guarantees linear runtime in the number of nodes of the subtree of the context node.

Time Complexity of Matching Backtracking Patterns

Due to the different semantics of backtracking patterns a differentiated analysis of their complexity is required. Let us first assume that backtracking pattern comprise only non-backtracking patterns as their inner patterns. In this case option and union patterns require a double time for matching in comparison to their inner patterns, because to decide whether they match two variants in general have to be examined. Hence their time complexity still will lie in $O(n)$ in the general case and $O(log\ n)$ in case of linguistically annotated documents.

Matching wildcard in a sequence (otherwise occurrences of wildcard are not reasonable and not used in the practice) requires in the average case backtracking on the half and in the worst case on the whole subtree of the context node resulting in

$$\sum_{i=0}^{n} 1 + T(n-i) = n + \sum_{i=0}^{n} T(i) \quad \text{or} \quad \sum_{i=0}^{\frac{n}{2}} 1 + T(n-i) = \frac{n}{2} + \sum_{i=\frac{n}{2}}^{n} T(i)$$

respectively, where $T(i)$ is the time complexity of matching the rest sequence after the wildcard. Assuming that the sequence does not contain any backtracking patterns and hence $T(s)$ to be linear the sequence with a wildcard requires $O(n^2)$ in average and worst cases. Our stack-based model helps to reduce the complexity saving a linear factor when evaluating different backtracking branches of the wildcard: while both stacks reflect the backtracking state after matching wildcard with the sequence of i nodes, matching the wildcard with the sequences of $i+1$ nodes corresponds to the removal of $i+1$-th node from the element stack, which requires a constant time (cf. the addition of 1 to $T(n)$ in the formulas above).

Since in the worst case checking of all permutations of the inner patterns is required, time complexity of matching permutation pattern is even exponential in the number of nodes. However, the asymptotic exponential runtime is quite irrelevant in practice because it effects may be precluded limiting the number of inner elements of permutation to some constant value, so that the absolute runtime does not significantly exceed that of other backtracking patterns.

Matching negation as a standalone pattern requires merely the time necessary for the matching of its inner pattern. However, negation as a part of a complex pattern has a serious impact on the time needed for its match as we will see in the next section.

Asymptotic Runtime of Matching Complex Recursive Patterns

Assessment of the runtime of matching complex patterns that consist of sequences of backtracking patterns that in turn comprise backtracking patterns as their inner patterns is a very difficult task because a very differentiated analysis is required and many assumptions about the kind of combinations, nesting degree of patterns and structure of the subtree of the context node have to be made. For example, while during the matching of combinations of backtracking patterns without negation complete backtracking is often unnecessary because the match is found in one of the backtracking branches, negation forces the evaluation of all backtracking branches containing the negation to ensure that the negated pattern does not occur in any branch (cf. sec. 6.3.4). The presence of negation implicates therefore worst case runtime.

Another important fact is the nesting degree, i.e. how often a backtracking pattern is enclosed by another backtracking pattern (e.g. consider the pattern `((("A" | "B")? * "C")%)*`). The time complexity of matching such complex nested patterns can be estimated multiplying the complexities of single backtracking patterns taking into account that in contrast to non-backtracking pattern the problem size remains constant for all nested backtracking pattern because every backtracking pattern independent of the nesting level will be matched on the subtree of the same context node (i.e. in our example every backtracking branch of the inmost union pattern begins on the same context node as the backtracking branches of the outer Kleene star pattern). However, if a backtracking pattern is nested by a non-backtracking pattern, e.g. `\NP[* "new" "version"]`, the problem size is reduced to the size of the subtree of the node that matches the non-backtracking pattern (i.e. element NP in our example).

Not all possible nesting combinations in complex patterns are reasonable and lead to the multiplication of the time complexity, since the resulting patterns may be simplified by the algebraic transformations, e.g. `(*)*=*` or `("A"?)*=("A")*` (cf. appendix C).

Since the runtime for matching of complex patterns depends on many parameters (number and kind of backtracking patterns, their nestings or sequences, presence of negation, number of non-backtracking patterns etc.) many assumptions have to be made to formally derive the asymptotic runtime in the average case. This average case would represent just one point in the huge spectrum of possible cases so that the assessment of time complexity for the general case of complex patterns is neither relevant nor reasonable. In our concrete scenario of matching linguistic patterns with linguistically preprocessed documents we can guarantee polynomial asymptotic runtime for the worst case (limiting the number of permutation elements to 5)

$$O(n^{2k} * (log\ n)^l * \frac{5!}{2} * m * n)$$

where k corresponds to the number of wildcards and Kleene Star patterns, l denotes the number of options and unions and m – the number of permutations. The worst case assumes that the backtracking patterns are nested by each other and matched with the top level context node, so that for most real patterns the runtime will be much shorter.

Above we have presented the analysis of time complexity for the general case. The real runtime on the preprocessed documents is usually lower than the theoretically derived asymptotic runtime because of optimizations of matching process exploiting the properties of the documents such as high locality of XML elements. Optimizations aim at reduction of backtracking cost by evaluating the backtracking branches first that are more likely to contain the match.

6.5 Summary

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.

Viewing patterns as XML queries opens a fully different view on our approach to IE. It can be regarded as a query-based approach in that information is identified and extracted by XML queries that are induced during the learning phase. Our IE system actually learns effective XML queries that retrieve relevant information from the linguistically preprocessed XML documents.