CLaRK — an XML-based System for Corpora Development*

Kiril Simov, Alexander Simov, Hristo Ganev, Milen Kouylekov,
Ilko Grigorov, Krasimira Ivanova.
BulTreeBank Project
http://www.BulTreeBank.org
Linguistic Modelling Laboratory, Bulgarian Academy of Sciences
Acad. G. Bonchev St. 25A, 1113 Sofia, Bulgaria
kivs@bultreebank.org, alex@bultreebank.org,
hristo@bultreebank.org, milen@bultreebank.org,
ilko@bultreebank.org, krassy_v@bultreebank.org

BulTreeBank Technical Report BTB-TR06

31.08.2004

1 Introduction

This report will present the main body of the documentation for the CLaRK System. It is based on a paper presented at the Corpus Linguistics 2001 conference in Lancaster. The idea of the paper is to present the main technologies on which the CLaRK System is based.

In this paper we describe the architecture and the intended applications of the CLaRK System. The development of the CLaRK System started under the Tübingen-Sofia International Graduate Programme in Computational Linguistics and Represented Knowledge (CLaRK). The main aim behind the design of the system is the minimization of human intervention during the creation of corpora. Creation of corpora is still an important task for the majority of languages like Bulgarian, where the invested effort in such development is very modest in comparison with more intensively studied languages like English, German and French. We consider the corpora creation task to be the editing, manipulation, searching and transforming of documents. Some of these tasks will be done for a single document or a set of documents, others will be done on a part of a document. Besides efficiency of the corresponding processing in each state of the work, the most important investment is the human labour. Thus, in our view, the design of the system has to be directed to minimization of the human work. For document management, storing and querying we chose the XML technology because of its popularity, flexibility and ease for understanding. It has become a part of our lives as it is the predominant language for data description and exchange on the Internet. Moreover, a lot of already developed standards for corpus descriptions like [XCES, 2001] and [TEI, 2001] are already adapted to the XML requirements. The core of the CLaRK System is an XML editor which is the main interface to the system. With the help of the editor the user can create, edit or browse XML documents. To facilitate the corpus management, we enlarge the XML inventory with facilities that support linguistic work. We added the following basic language processing modules: a tokenizer with a module that supports a hierarchy of token types, a finite-state engine that supports the writing of cascaded finite-state grammars and facilities that search for regular expression patterns, the XPath query language which is able to support navigation over the whole set of mark-up

*This work is funded by the Volkswagen Stiftung, Federal Republic of Germany under the Programme "Cooperation with Natural and Engineering Scientists in Central and Eastern Europe" contract I/76 887.

of a document, mechanisms for imposing constraints over XML documents which are applicable in the context of some events. We envisage several uses for our system:

1. *Corpora markup.* Here users work with the XML tools of the system in order to mark-up texts with respect to an XML DTD. This task usually requires an enormous human effort and comprises both the mark-up itself and its validation afterwards. Using the available grammar resources such as morphological analyzers or partial grammars, the system can state local constraints reflecting the characteristics of a particular kind of texts or mark-up. One example of such constraints can be as follows: a PP according to a DTD can have as parent a NP or VP, but if the left sister is a VP then the only possible parent is VP. The system can use such kind of constraints in order to support the user and minimize his/her work.

2. *Dictionary compilation for human users.* The system will support the creation of the actual lexical entries whose structure will be defined via an appropriate DTD. The XML tools will be used also for corpus investigation that provides appropriate examples of the word usage in the available corpora. The constraints incorporated in the system will be used for writing a grammar of the sublanguages of the definitions of the lexical items, for imposing constraints over elements of lexical entries and the dictionary as a whole.

3. *Corpora investigation.* The CLaRK System offers a rich set of tools for searching over tokens and mark-up in XML corpora, including cascaded grammars, XPath language. Their combinations are used for tasks such as: extraction of elements from a corpus - for example, extraction of all NPs in the corpus; concordance - for example, give me all NPs in the context of their use ordered by a user defined set of criteria.

The structure of the paper is as follows: in the next section we give a short introduction to the main technologies on which the CLaRK System is built: these are: **XML technology** - notions of XML language and XPath querying language; **Cascaded regular grammars** - regular expressions, cascaded grammars, application within the XML technology; **Unicode-based tokenizers** - the encoding within the system is based on Unicode and the user can define a hierarchy of tokenizers depending on the needs; **Constraint engines** - several techniques for imposing restrictions over documents' content on the basis of different conditions (for example, the context); **MultiQuery tool** - an architecture for integration of different system tools in complex processing procedures. The third section describes the main components of the CLaRK System and their functionality, the last section outlines some directions for future development.

2 The technologies behind the CLaRK System

2.1 XML technology

2.1.1 eXtendible Markup Language (XML)

XML stands for *eXtendible Markup Language* (see [XML, 2000]) and it emerged as a new generation language for data description and exchange for Internet use. The language is more powerful than HTML and easier to implement than SGML. Starting as a markup language, XML is evolved into a technology for structured data representation, exchange, manipulation, transformation, and querying. The popularity of XML and its ease for learning and use made it a natural choice for the basis of the CLaRK System. This section presents in an informal way some of the most important notions of the XML technology. For more rigorous and full presentation the reader is directed to the corresponding literature on the following address: `http://www.w3c.org/XML/`. XML defines the notion of structured document in terms of sequences and inclusions of elements in the structure of the document. The whole document is considered as an element which contains the rest of the elements. The elements of the structure of a document are marked-up by means of *tags*. Tags can surround the *content* of an element or tags can mark

some points in the document. In the first case the beginning of an element is marked-up by the so called opening tag written as `<tagname>` and the end is marked-up by a closing tag written as `</tagname>`. For example, TEI documents include on top level the following two elements:

```
<TEI.2>
    <TeiHeader> ... content of the TEI header element ...
    </TeiHeader>
    <text> ... Content of the text element ... </text>
</TEI.2>
```

The tags of the second kind, so-called empty elements, are usually represented as `<tag/>`. For example, a line break within a sentence can be represented in the following way:

```
<s>...first line of text... <lb/> ...second line of text...</s>
```

Each element can be connected with a set of *attributes* and their *values*. The currently assigned set of attributes of an element is recorded within the opening tag of the element or before the closing slash in an empty element. Some of the attribute-value pairs are by default assigned to some tags and thus it is not obligatory to list them.

One important requirement for an XML document is that elements having common content must strictly include one into another. This means that overlap of the elements is forbidden. Such documents are called *well-formed*. For example, the following document *is not well-formed* and thus it is not an acceptable XML document:

```
<doc><el1> ... <el2> ... </el1> ... </el2></doc>
```

The XML technology defines a set of mechanisms for imposing constraints over XML documents. Such kind of a very basic mechanism is the so called DTD (Document Type Definition). A DTD defines the inclusion of elements and the possible sequences of elements within the content of an element. These definitions are given as ELEMENT statements in the DTD. Each ELEMENT statement has the following format:

```
<!ELEMENT tagname content_definition>
```

where *tagname* is the name of the element and *content_definition* is a definition of the content of this kind of elements. Besides some reserved words as `EMPTY` and `ANY`, the content definition is represented as a regular expression over tag names. This regular expression determines the tags and their order in the content of the enclosing element. Additionally, the DTD can contain definitions of the allowed attributes for the elements, entity declarations and others. For more details, the interested reader is directed to the literature on the corresponding notions - see the above address.

An XML document containing elements whose content obeys the restrictions stated in a DTD is said to be *valid* with respect to this DTD.

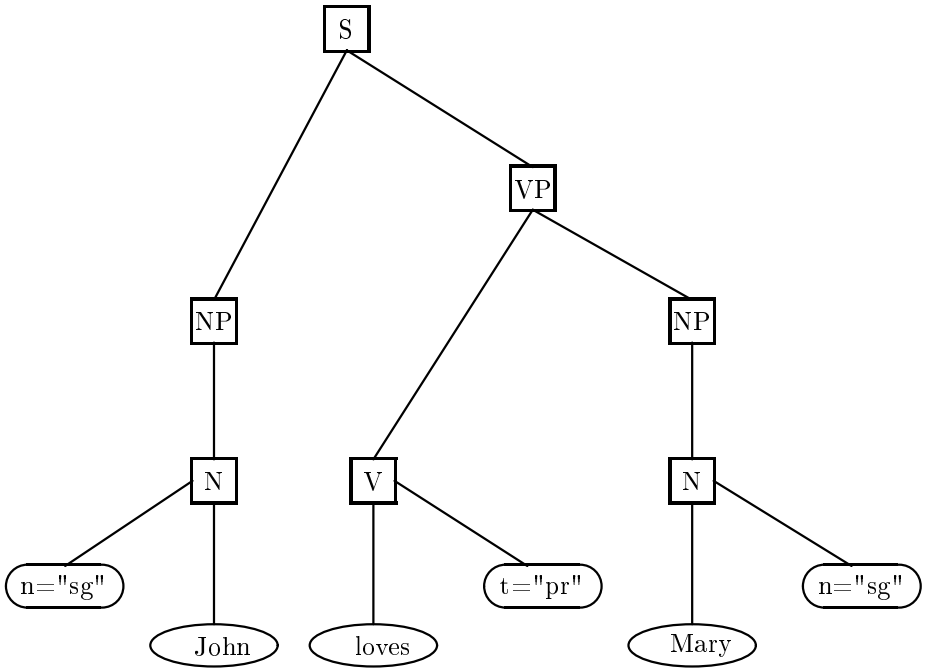
2.1.2 XML Path Language (XPath)

Another important language defined within the XML world and used within the CLaRK System is the XPath language. XPath is a powerful language for selecting elements from an XML document. The XPath engine considers each XML document as a tree where the nodes of the tree represent the elements of the document, the document's outer most tag is the root of the tree and the children of a node represent the content of the corresponding element. Attributes and their values of each element are represented as an addition to the tree. This section is based on the following document [XPath, 1999].

For instance, the following XML document:

```
<S>
 <NP><N n="sg"> John </N></NP>
 <VP>
         <V t="pr"> loves </V>
         <NP><N n="sg"> Mary </N></NP>
 </VP>
</S>
```

is represented as the following tree:



In this picture the rectangles correspond to elements in the XML document and they are called *element nodes*, the ovals represent the text elements in the document and they are called *text nodes*, and the rounded rectangles contain the attribute-value pairs and they are called *attribute nodes*. Each rectangle contains the tag of the corresponding element, each text node contains the text from the content of the corresponding element in the document. Each attribute node is attached to the element that contains it. The root of the tree corresponds to the element containing the whole document. The immediate descendant nodes of a given node N represent the content of the node N. The attribute nodes are considered in a different way and they are connected with the corresponding nodes in a different dimension. Thus the attribute nodes are not descendant nodes of any node in the tree.

The XPath language uses the tree-based terminology to point into some direction within the tree. One of the basic notions of the XPath language is the so called *context node*, i.e. a chosen node in the tree. Each expression in the XPath language is evaluated with respect to some context node. The nodes in the tree are categorized with respect to the context node as follows: the nodes immediately under the context node are called *children* of the context node; all nodes under the context node are called *descendant* nodes of the context node; the node immediately above the context node is called *parent* of the context node; all nodes that are above the context node are called *ancestor* nodes of the context node; the nodes that have the same parent as the context node are called *sibling* nodes of the context node; siblings of the context node are divided into two types: *preceding siblings* and *following siblings*, depending on their order with respect to the context node in the content of their parent - if the sibling is before the context

node, then it is a preceding sibling, otherwise it is a following sibling. Attribute nodes are connected to the context node as *attribute* nodes and they are not children, descendant, parent or ancestor nodes of the context node. The context node with respect to itself is called *self* node. Some examples: let the `"VP"` node in the above tree be the context node, then its children are the `"V"` node and `"NP"` node that immediately dominates the node `"Mary"`; its descendant nodes (or simply descendants) are the nodes: `"V"`, `"loves"`, `"NP"`, `"N"`, `"Mary"`; its parent is the node `"S"`; this node is also an ancestor of the node `"VP"`; its sibling is the left node `"NP"` and more precisely this is a preceding sibling. If the context node is `"V"` then its ancestor nodes (or simply ancestors) are the nodes `"VP"` and `"S"`; its attribute node is the node `'t="pr"'`; its following sibling is the right `"NP"` node.

Another important concept of the XPath language is the *location path*. Informally, the location path is a description of how to reach some nodes in the tree from the context node. More formally, the location path is a kind of expression. The location path selects a set of nodes relative to the context node. The result of evaluating a location path expression is a node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. The next section describes the syntax and meaning of location path expressions.

Location Paths

Although location paths are not the most general grammatical construct in the language, they are the most important construct and therefore they will be described first. Every location path can be expressed via a straightforward but rather verbose syntax. There are also a number of syntactic abbreviations specific for the CLaRK System (not 100% conformant to the recommendation) that allow common cases to be expressed concisely. This section will explain the semantics of location paths using the unabbreviated syntax. The abbreviated syntax will then be explained by showing its expansion into the unabbreviated syntax . Here are some examples of location paths using the unabbreviated syntax:

- `child::para` selects the `para` element children of the context node
- `child::*` selects all children (element and text nodes) of the context node (different from the recommendation)
- `child::text()` selects all text node children of the context node
- `child::node()` selects all children (element and text nodes) of the context node (different from the recommendation)
- `attribute::name` selects the name attribute of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::para` selects the `para` element descendants of the context node
- `ancestor::div` selects all `div` ancestors of the context node
- `ancestor-or-self::div` selects the `div` ancestors of the context node and, if the context node is a `div` element, selects the context node itself as well
- `descendant-or-self::para` selects the `para` element descendants of the context node and, if the context node is a `para` element, selects the context node itself as well
- `self::para` selects the context node if it is a `para` element, and otherwise selects nothing
- `child::chapter/descendant::para` selects the `para` element descendants of the `chapter` element children of the context node
- `child::*/child::para` selects all `para` grandchildren of the context node
- `/descendant::para` selects all the `para` elements in the same document as the context node

- `/descendant::olist/child::item` selects all the `item` elements that have an `olist` parent and that are in the same document as the context node
- `child::para[position()=1]` selects the first `para` child of the context node
- `child::para[position()=last()]` selects the last `para` child of the context node
- `child::para[position()=last()-1]` selects the second last `para` child of the context node
- `child::para[position()>1]` selects all `para` children of the context node excluding the first `para` child of the context node
- `following-sibling::chapter[position()=1]` selects the next `chapter` sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the previous `chapter` sibling of the context node
- `/descendant::figure[position()=42]` selects the forty-second `figure` element in the document
- `/child::doc/child::chapter[position()=5]/child::section[position()=2]` selects the second `section` of the fifth `chapter` of the `doc` document element
- `child::para[attribute::type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `child::para[attribute::type='warning'][position()=5]` selects the fifth `para` child of the context node that has a `type` attribute with value warning
- `child::para[position()=5][attribute::type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning` ·
- `child::chapter[child::title='Introduction']` selects the `chapter` children of the context node that have one or more `title` children with *string-value* equal to `Introduction`
- `child::chapter[child::title]` selects the `chapter` children of the context node that have one or more `title` children
- `child::*[self::chapter or self::appendix]` selects the `chapter` and `appendix` children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last `chapter` or `appendix` child of the context node

There are two kinds of location path: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by / (slash). The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are united together. The set of nodes identified by the composition of the steps is this union. For example, `child::div/child::para` selects the `para` element children of the `div` element children of the context node, or, in other words, the `para` element grandchildren that have `div` parents.

An absolute location path consists of / followed by a relative location path. The slash / selects the root node of the document as a context node for the following relative location path. The slash itself is not a location path in the CLaRK System. If the root node has to be selected, the expression `/self::*` is to be used.

Location Steps

A location step has three parts:

- an axis, which specifies the tree relationship between the nodes selected by the location step and the context node,
- a node test, which specifies the node type or the name of the nodes selected by the location step, and
- zero or more predicates, which use arbitrary expressions for further refining the set of nodes selected by the location step.

The syntax for a location step is the axis name and node test separated by two colons, followed by zero or more expressions, each in square brackets. For example, in `child::para[position()=1]`, *child* is the name of the axis, *para* is the node test and *[position()=1]* is a predicate.

The node-set selected by the location step is the node-set that results from generating an initial node-set from the axis and node-test, and then filtering that node-set by each of the predicates in turn.

The initial node-set consists of the nodes having the relationship to the context node specified by the axis, and having the node type or name specified by the node test. For example, a location step `descendant::para` selects the `para` element descendants of the context node: `descendant` specifies that each node in the initial node-set must be a descendant of the context; `para` specifies that each node in the initial node-set must be an element named `para`. The available axes are described in **Axes**. The available node tests are described in **Node Tests**.

The initial node-set is filtered by the first predicate to generate a new node-set; this new node-set is then filtered using the second predicate, and so on. The final node-set is the node-set selected by the location step. The axis affects the evaluation of the expression in each predicate and so the semantics of a predicate is defined with respect to an axis.

Axes

The following axes are available in the implementation of XPath in the CLaRK System:

- the **child** axis contains the children of the context node
- the **descendant** axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes
- the **parent** axis contains the parent of the context node, if there is one
- the **ancestor** axis contains the ancestors of the context node; the ancestors of the context node consist of the *parent* of the context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node
- the **following-sibling** axis contains all the following siblings of the context node; if the context node is an attribute node, the `following-sibling` axis is empty
- the **preceding-sibling** axis contains all the preceding siblings of the context node; if the context node is an attribute node, the `preceding-sibling` axis is empty
- the **following** axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and attribute nodes
- the **preceding** axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and attribute nodes
- the **attribute** axis contains the attributes of the context node; the axis will be empty unless the context node is an element
- the **self** axis contains just the context node itself
- the **descendant-or-self** axis contains the context node and the descendants of the context node

- the **ancestor-or-self** axis contains the context node and the ancestors of the context node; thus, the `ancestor-or-self` axis will always include the root node
- the **document** axis is used to refer an external document. It contains the root node of the external document. This axis is always used with the name test *n("document_name")* which specifies the external document name. For example: document::n("dict.xml") will select the root node of a document, named *dict.xml* [1].
- the **next** and the **previous** axes contain the first following or the first preceding sibling if it satisfies the subsequent node/name test.
  [2]

NOTE: The `ancestor`, `descendant`, `following`, `preceding` and `self` axes partition the document (ignoring attributes): they do not overlap and together they contain all the nodes in the document.

Node Tests

The node tests are divided into two categories: node type tests and node name tests.

Here are the node type tests implemented in the system:

- `text()` - from the initial node-set it preserves only the text nodes.
- `text(<text>)` - from the initial node-set it preserves only the text nodes, which contain the `<text>` as a substring of their text content. For example `child::text("play")` will return all text nodes which are children of the context node and contain the token "play" in them. This is specific for the CLaRK System[3].

  `<text>` in each of the above uses is, in fact, a token description and could include wildcard symbols as it is described on page 21.
- `node()` - this node test does not filter the initial node-set. It is used when all the nodes selected from the axis are needed for further evaluation. For short, `"*"` can be used, i.e. `child::*` instead of `child::node()`. This is specific for the CLaRK System. ·
- `element()` - from the initial node-set only the element nodes remain.
- `attribute()` - from the initial node-set only the attribute nodes remain. It is possible for the initial nodes to contain not only attributes.
- `attribute(<attributeName>)` Filters only for element nodes which have an attribute named `<attributeName>`. Example: `child::attribute(id)` will take only the element nodes, children of the context node, which have an attribute id. This is specific for the CLaRK System.
- `attribute(<attributeName> = "<attributeValue>")` This node test is almost the same as the preceding node test `attribute(<attributeName>)`, but in addition it also has a restriction on the value of the attribute. Example: `child::attribute(id="243")` will take only these element nodes which have an attribute `id` set to value 243. This is specific for the CLaRK System.

[1] An extension of the language allowing cross document references.

[2] An extension of the language aiming to speed up the process of evaluation for such specific cases.

[3] Additional possibilities in the system are:

`text(<mode>,<text>)`

From the initial node-set preserves only the text nodes, which contain the `<text>` in their text content. Four modes are available. Mode 1 - The text node content is starting with the `<text>`. Mode 2 - The text node contains the `<text>`. Mode 3 - The text node content ends with the text. Mode 4 The text node content is the same as the `<text>`. For example `child::text(3,"play")` will return all text nodes which are children of the context node and contain the token "play" in the end of their content.

`text(<mode>,<y|n>, <"(" regular expression ")">)`

From the initial node-set preserves only the text nodes, which contain text that matches the `< regular expression >`. The text is tokenized by the tokenizer for the element parent of this text node or by the default tokenizer for the DTD if the first is not presented. The second parameter is for normalization. The user must select either `"y"` (yes) or `"n"` (no). The modes are equivalent to the previous node test. For example `child::text(3,y,"play")` will return all text nodes which are children of the context node and contain the token "play" or "Play" or "pLAY" or ... in the end of their content.

- `comment()` - from the initial node-set only comment nodes remain.
- `processing-instruction()` - from the initial node-set only processing-instruction nodes remain.

The name node tests are used to filter the initial node-set for element nodes with a given name. All other non-element nodes and element nodes with other names are removed from the set. For example, `child::para` takes only the *para* element node children of the context node.

Predicates

An axis is either a forward axis, or a reverse axis. An axis that only contains the context node or nodes that follow the context node in *document order* is a forward axis. An axis that only contains the context node or nodes that precede the context node in *document order* is a reverse axis. Thus, the `ancestor`, `ancestor-or-self`, `preceding` and `preceding-sibling` axes are reverse axes; all other axes are forward axes. Since the `self` axis always contains only one node, it makes no difference whether it is a forward or reverse axis. The `proximity position` of a member of a node-set with respect to an axis is defined in the following way: the position of the node in the node-set is ordered in document order if the axis is a forward axis and it is ordered in reverse document order if the axis is a reverse axis. The first position is 1.

A predicate filters a node-set with respect to an axis in order to produce a new node-set. For each node in the node-set which is to be filtered, the predicate is evaluated with the node in question as the context node, with the number of nodes in the node-set as the context size, and with the *proximity position* of the node in the node-set with respect to the axis as the context position; if the predicate evaluates to true for that node, the node is included in the new node-set; otherwise, it is excluded.

The predicate is evaluated as an expression and the result is converted to a boolean. If the result is a number, the result will be converted to true if the number is equal to the proximity position of the currently tested node and will be converted to false otherwise; if the result is not a number, then the result will be converted as if by a call to the `boolean()` function. Thus a location path `para[3]` is equivalent to `para[position()=3]`. In other words if the context node has 4 child nodes *para*, then only for the third *para* child the predicate `[3]` (or `[position()=3]`) will be true. So the new node-set will contain only the third *para* child.

Variable Bindings

One significant extension of the XPath language implemented in the system is the ability partial results to be stored during evaluation time and used later in other expressions evaluations. This feature can be used to optimise the processing time in expressions where sub expressions are evaluated many times under same conditions. Another possibility is changing the current context by saving it in a variable, doing some other computations and then restoring the context. A typical usage of this is extracting some data from an external document on the basis of local context information (for example, a dictionary look-up) and then proceeding the evaluation within the context The syntactic construction of the variable binding is the following: **("{", *variable_name*, ":=", *var_assignment_expr*,"}")***, *main_expr*. Here the `variable_name` is an arbitrary string containing letters, `var_assignment_expr` and `main_expr` are valid XPath expressions. In the process of evaluation the `var_assignment_expr` is processed first, the result value is assigned to the variable and then the `main_expr` is processed. The scope of usage of the variable value is within the following variable definition expressions (if there are such) and within the `main_expr`. Variable assignments can be used at any level of sub expressions. A variable value can have any type of the standard ones: a node-set, a number, a string or a boolean.

Abbreviated Syntax

Here are some examples of location paths using abbreviated syntax:

- `para` selects the `para` element children of the context node
- `*` selects all children of the context node
- `text()` selects all text nodes, children of the context node

- `@name` selects the `name` attribute of the context node
- `@*` selects all the attributes of the context node
- `para[1]` selects the first `para` child of the context node
- `para[last()]` selects the last `para` child of the context node
- `*/para` selects all `para` grandchildren of the context node
- `/doc/chapter[5]/section[2]` selects the second `section` of the fifth `chapter` of the `doc`
- `chapter//para` selects the `para` element descendants of the `chapter` element children of the context node
- `//para` selects all the `para` descendants of the document root and thus selects all `para` elements in the same document as the context node
- `//olist/item` selects all the `item` elements in the same document as the context node that have an `olist` parent
- `.` selects the context node
- `.//para` selects the `para` element descendants of the context node
- `..` selects the parent of the context node
- `../@lang` selects the `lang` attribute of the parent of the context node
- `para[@type="warning"]` selects all `para` children of the context node that have a `type` attribute with value `warning`
- `para[@type="warning"][5]` selects the fifth `para` child of the context node that has a `type` attribute with value `warning`
- `para[5][@type="warning"]` selects the fifth `para` child of the context node if that child has a `type` attribute with value `warning`
- `chapter[title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with *string-value* equal to `Introduction`
- `chapter[title]` selects the `chapter` children of the context node that have one or more `title` children
- `employee[@secretary and @assistant]` selects all the `employee` children of the context node that have both a `secretary` attribute and an `assistant` attribute

The most important abbreviation is that `child::` can be omitted from a location step. In effect, `child` is the default axis. For example, the location path `div/para` is short for `child::div/child::para`.

There is also an abbreviation for attributes: `attribute::` can be abbreviated to `@`. For example, a location path `para[@type="warning"]` is short for `child::para[attribute::type="warning"]` and therefore it selects `para` children with a `type` attribute with value equal to `warning`.

`//` is short for `/descendant-or-self::node()/`.

For example, `//para` is short for `/descendant::para` and so will select any `para` element in the document; `div//para` is short for `div/descendant::para` and therefore it will select all `para` descendants of `div` children.

A location step of `.` is short for `self::node()`. This is particularly useful in conjunction with `//`. For example, the location path `.//para` is short for `self::node()/descendant::para` and therefore it will select all `para` descendant elements of the context node.

Similarly, a location step of `..` is short for `parent::node()`.

For example, `../title` is short for `parent::node()/child::title` and therefore it will select the `title` children of the parent of the context node.

2.1.3 XSL Transformations (XSLT)

Here the main ideas about XSLT are described and a list of the implemented and unimplemented features is given.

XSLT offers a way to transform a document by applying transformation rules. For complete information about the XSLT standard see ([XSLT, 1999]). Not everything stated in the standard is implemented in the CLaRK System, and some things work differently. For the application of a transformation two things are necessary:

1. A source document that can be any well-formed XML document.
2. A transformation document (TD) that contains the transformation rules.

There are two modes of application of a transformation. The first one is the classical one where the result from the transformation is a new document. It is created on the basis of a source XML document and a transformation rules applied on it. The second mode of application is a system specific one where the transformation involves modification of the source document which represents the final result. This mode is called a `Local XSLT Transformation`. The main point in the second mode is that certain points are selected in the source document and the transformation is applied locally for each of them as if it is a root of a document. The result from each single transformation substitutes the corresponding selection point. When a transformation gives no result, the corresponding node (selection point) is deleted from the source. The initial selection points are determined by an XPath expression evaluation. In the process of transforming the user is given the possibility to apply the transformation to all selected points at once, or to go through each of them one by one and to confirm/reject each single application.

The rest of this section will be concerned with the transformation document. In the CLaRK System every transformation document is an XML document.

Basic Structure of the Transformation Document

As we said each transformation document in the CLaRK System is a well-formed XML document which can be edited in the system. Some of the tag names used in transformation documents are reserved tag names defined in the XSL standard. In order they to be separate from the user-defined tag names, they usually may be preceded by an *xsl* prefix followed by a colon (:) (so a tag named *template* may be also written as *xsl:template*) or in other sense a namespace prefix for XSL. In the following paragraphs the syntax of the TD is given. For details of how the XSLT processor work see the next section.

The root

The root element of every TD must be marked with either the *transform* or *stylesheet* tag.

Children of the root

Child elements of the root element are marked with a *template* tag. Each of them contains exactly one template rule.

Templates

Attributes of the *template* tag

match attribute

The `match` attribute contains a string that is interpreted as an XPath expression. The pattern matches a node A in the source document iff there exists a node C in the source document such that when the XPath expression in the match attribute is evaluated with C as a context node, the result is a node list and A is member of this list.

mode attribute

The `mode` attribute contains a string that determines the mode in which the XSLT processor must be in order to use this rule. The string is completely arbitrary. The default is the empty string.

priority attribute.

The `priority` attribute determines the priority of the rule. Priorities are used to determine which rule to use if more than one rule matches a given node. If no value is given for this attribute, 0 is substituted. If more than one template still remain, the last defined template is used.

Children of the `template` tag

The children elements of a template element could be of any kind, but some tag names have special meaning.

`apply-template` tag

This tag tells the XSLT processor to continue applying the rules.

select attribute

This attribute contains a XPath expression, that is evaluated with the node that the template has matched as a context node. Then each node in the resulting node set is processed. If no value is given, `child::*` is assumed.

mode attribute

This attribute sets the XSLT processor in the given mode. The `apply-template` tag can have optional `sort` children.

`for-each` tag

This tag applies its content to every node in the node set it selects. Its content is similar in every way to the content of a `template` tag and is similarly used, but it may have `sort` tag children that determine the order of processing of the nodes.

Attributes:

select attribute

A XPath that selects the nodes to be processed with the body of the `for-each` tag.

`sort` tag

The `sort` tag determines the key for sorting of the lists, resulting from XPath expression. When no such tag occurs, the list is processed in document order. Otherwise, it is first sorted. The first sort child determines the first key, the second sort child the second key and so on.

select attribute

This attribute holds an XPath expression which is evaluated for every node in the list and is then converted either to string or to number. It determines the key value. The default value is `"."`.

data-type attribute

This attribute determines whether the key is converted to a string or to a number. It can have the values a `string` or a `number`. The default is `string`.

order attribute

Specifies the order of the resulting list. Valid values are `ascending` and `descending`. Default is `ascending`.

The `sort` tags may appear as children of `for-each` and `apply-templates` tags. Elsewhere it is considered as a normal tag without special meaning.

`if` tag

This tag contains an XPath expression that is evaluated as a boolean. If the answer is true, then the processor applies the content of the tag to the current node. Otherwise it does nothing.

test attribute

The XPath expression that is evaluated to boolean. It behaves exactly as if the XPath expression was the following expression `boolean(original_expression)`.

`copy-of` **tag**

This tag contains an XPath expression, that is evaluated either to a list of nodes, or to a boolean, to a number or a string. In the first case each of the selected nodes together with its subtree is attached to the result. In the other cases the XPath is converted to string and is then added to the result tree.

select attribute

Selects the nodes (or data) to be copied.

`copy` **tag**

This tags copies the current node and its attributes. The subtree is not copied. No attributes are defined for this tag.

`value-of` **tag**

This tag behaves like `copy-of` tag with `select` attribute converted to `string(original_expression)`.

select attribute

Selects the nodes (or data) to be converted and copied.

These are the special meaning tags. Every other tag or character data is directly copied to the result tree.

XSLT processor

The XSLT processor takes as input a source document (SD) and a transformation document (TD) and returns a new document that is called the result document (RD). Its works as follows :

1. Finds the template that matches the document element of the SD and has mode "" and highest priority.
2. Applies that template.

In the template there may be "apply-template" tags that run the processor recursively. Every node that has no special meaning is attached to the result tree in the right position node (RP). At first this RP is the document node of the RD itself. For more information see the examples below.

As one may see, the resulting document may not have only one root. To avoid this, only the first child of the Document node is actually used. This is not the case when you apply the transformation inside the document rather than to the entire document.

Examples

The “books” example.

Lets assume that we have a set of “books”, and every “book” tag has zero or more “author” tags. We want to extract all the authors and put them in a new document. The following XSLT document will do the job.

Example:

```
<books>
  <book><title>Alice in Wonderland</title>
              <author>Lewis Carrol</author></book>
  <book><title>Creatures of Light and Darkness</title>
              <author>Roger Zelazny</author></book>
  <book><title>The Chronicles of Amber </title>
              <author>Roger Zelazny</author></book>
</books>
```

Note: We will use the "//" for comments. They are not part of the TD.

```
<transform> <template match="*"> // the only template that matches every tag
      <authors>    // this tag has no special meaning
        <for-each select="child::book/child::author">
                    // for every author of every book
                <sort/> // sort the authors
                <copy-of select="."/> // copy its "author"s tags
        </for-each> // end cycle
      </authors>
</template> // end template
</transform> // end the TD
```

The result will be

```
<authors>
 <author>Lewis Carrol</author>
 <author>Osenova</author>
 <author>Popova</author>
 <author>Roger Zelazny</author>
 <author>Roger Zelazny</author>
 <author>Simov</author>
</authors>
```

As one may see, we will have authors that are mentioned more than once in the RD. Lets fix this! The new SD will be the old RD, and the transformation is this :

```
<transform>
 <template match="authors"> // match the root
   <copy> // copy the root
     <apply-templates/> // apply-templates to its children
   </copy>
 </template>
 <template match="author"> // match every author
   <if test="(. != following-sibling::author[1])">
               // check if it's the same as its next sibling
               // note that the authors are sorted
       <copy-of select="."/>
   </if>
 </template>
</transform>
```

And the final result is

```
<authors>
 <author>Lewis Carrol</author>
 <author>Osenova</author>
 <author>Popova</author>
 <author>Roger Zelazny</author>
 <author>Simov</author>
</auto hrs>
```

Besides XSLT, the CLaRK System also provides additional means for transformations of a document via the XPath Transformation tool. It is described in another part of the documentation.

2.2 Cascaded Regular Grammars

The CLaRK System is equipped with a finite-state engine which is used for several tasks in the system. In this section we present the use of this engine for cascaded regular grammars over XML documents along the lines described in [Abney, 1996]. The general idea underlaying cascaded regular grammars is that there is a set of regular grammars. The grammars in the set are in particular order. The input of a given grammar in the set is either the input string if the grammar is first in the order or the output string of the previous grammar. Another feature of cascaded grammars is that each grammar tries to recognise only a particular category in the string but not the whole string. Before going into details of how to apply grammars in the CLaRK System some basic notions about regular expressions are given.

2.2.1 Regular Expressions and Grammars

This is not a completely formal presentation of the topic. Our purpose here is to clarify the terminology that will be used further in the document.

Letter

We call a *letter* each designate symbol. Symbol here is understood in a general way and not as a symbol code in the computer. Symbol can be a word from a natural language as 'love' in English.

Alphabet

An *alphabet* is a set of letters.

Word

A *word* is a finite sequence of letters over some alphabet. The empty sequence of letters will be called the empty word and we will write it as ϵ. The set of all words over the alphabet A is denoted by $A*$. A basic operation over words is *concatenation* – $\oplus$. The concatenation is a binary operation. The concatenation of two words is the word which consist of the letters of the first word followed by the letters of the second word. For example:

$$grand \oplus mother = grandmother$$

Sometimes we simplify the notation and we write **l** for the letter **l** and for the word that consists of this letter.

Language

A *language* is a set of words over some alphabet.

Regular expression

A *regular expression* over an alphabet is an expression that can be interpreted as a set of words over this alphabet. This set of words is the language that is recognised by the regular expression. Before giving the syntax and semantics of the regular expressions we will enrich our formal inventory with *letter descriptions*. A letter description is an expression ld which denotes a set of letters from a given alphabet. Each letter from a given alphabet is a description of itself. The set of letters denoted by the letter description ld is represented as $A(ld)$, i.e. `[a-z]`, `\p`.

Syntax of the regular expressions:

1. (Basic case)	If ld is a letter description, then ld is a regular expression.
2. (Concatenation)	If r_1 and r_2 are regular expressions, then (r_1, r_2) is a regular expression.
3. (Union)	If r_1 and r_2 are regular expressions, then $(r_1 \mid r_2)$ is a regular expression.
4. (Intersection)	If r_1 and r_2 are regular expressions, then $(r_1 \ ! \ r_2)$ is a regular expression.
5. (Subtraction)	If r_1 and r_2 are regular expressions, then $(r_1 \ - \ r_2)$ is a regular expression.
6. (One or zero)	If r is a regular expression, then $r?$ is a regular expression.
7. (Zero or more)(Kleene star)	If r is a regular expression, then $r*$ is a regular expression.
8. (One or more)(Kleene plus)	If r is a regular expression, then $r+$ is a regular expression.
9. (Negation)	If r is a regular expression, then $\tilde{}r$ is a regular expression.

Semantics of the regular expressions:

The semantics of the regular expressions is defined by an interpretation function $L(.)$ which maps each regular expression into the set of words denoted by this regular expression. The function $L(.)$ is total and meets the following equations:

1. (Basic case)

   If ld is a letter description, then $L(ld) = \{l \mid l \in A(ld)\}$

Thus each letter description ld denotes the set of words corresponding to the letters described by ld.

2. (Concatenation)

   If r_1 and r_2 are regular expressions, then $L((r_1, r_2)) = \{w_1 \oplus w_2 \mid w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$

Thus the interpretation of a concatenation expression is the set of words so that each word in the set is a concatenation of two words, where the first word is from the interpretation of the first regular expression and the second one is from the interpretation of the second expression.

3. (Union)

   If r_1 and r_2 are regular expressions, then $L((r_1 \mid r_2)) = \{w \mid w \in L(r_1) \text{ or } w \in L(r_2)\}$

Thus the interpretation of a union expression is the set of words so that each word in the set is either from the interpretation of the first regular expression or from the interpretation of the second expression.

Another way to define the interpretation of a union expression is by $L((r_1 \mid r_2)) = L(r_1) \cup L(r_2)$.

4. (Intersection)

   If r_1 and r_2 are regular expressions, then $L((r_1 \mathbin{!} r_2)) = \{w \mid w \in L(r_1) \text{ and } w \in L(r_2)\}$

Thus the interpretation of an intersection expression is the set of words so that each word in the set is at the same time in the interpretations of the both regular expressions.

5. (Subtraction)

   If r_1 and r_2 are regular expressions, then $L((r_1 \ - \ r_2)) = \{w \mid w \in L(r_1) \text{ and } w \notin L(r_2)\}$

Thus the interpretation of a subtraction expression is the set of words so that each word in the set is in the interpretation of the first regular expression, but not in the interpretation of the second one.

6. (One or zero)

   If r is a regular expression, then $L(r?) = \{\epsilon\} \cup L(r)$.

Thus the interpretation of an one-or-zero expression is the interpretation of the expression united with the set of the empty word.

7. (Zero or more)(Kleene star)

   If r is a regular expression, then $L(r*) = \{\epsilon\} \cup \{w_1 \oplus w_2 \oplus \ldots w_n \mid w_I \in L(r) \text{ and } n > 0 \text{ is a natural number } \}$.

Thus the interpretation of a zero-or-more expression is the set of the empty word united with the set of all words in the interpretation of the expression r and all words that are concatenations of two or more words from the interpretation of the expression r.

8. (One or more)(Kleene plus)

   If r is a regular expression, then $L(r+) = \{w_1 \oplus w_2 \oplus \ldots w_n \mid w_I \in L(r) \text{ and } n > 0 \text{ is a natural number } \}$.

Thus the interpretation of an one-or-more expression is the set of all words in the interpretation of the expression r and all words that are concatenations of two or more words from the interpretation of the expression r.

9. (Negation)

If r is a regular expression, then $L(\tilde{}r) = \{w \mid w \notin L(r)\}$

Thus the interpretation of a negation expression is the set of all words which are NOT in the interpretation of the regular expression r.

Some simplifications of the syntax are allowed. When only the operations concatenation and union are used, the internal brackets can be omitted. For instance, $((a, b), (c, d))$ can be written as (a, b, c, d).

The set of words in the interpretation of a regular expression is called the language recognised by the expression. Each language that can be recognised by a regular expression is called a *regular language*. Notice that different regular expressions can recognise the same language. Also a word can belong to different languages.

Regular Grammars

A regular expression can be used to determine whether a given word belongs to the language recognised by the expression, but sometimes we want to recognise the internal structure of a word. This can be done by segmenting the word in subwords and by assigning some categories to these subwords. For instance, let we write a regular expression for a recognition of dates in the format `dd.mm.yyyy` (10.11.2002):

```
( ( 0, (1|2|3|4|5|6|7|8|9) ) | ( (1|2), (0|1|2|3|4|5|6|7|8|9) ) | (3,(0|1)) )
,
.
,
( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) )
,
.
,
(( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

This regular expression successfully recognises 10.11.2002 as a word over the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .}. It also accepts some words that are not dates as 31.02.1999, but this is not important in our example.

Sometimes we would like to segment the date in a day part, a month part and a year part. In order to do this we use rules of the following kind:

```
C -> R
```

where R is a regular expression and C is a category of the words recognised by `R`. We can think of `C` as a name of the language recognised by `R`.

A *regular grammar* is a set of rules.

A regular grammar works over a word (called input word) and it tries to segment this word in a sequence of subwords in such a way that each of these subwords in recognised by a regular expression of the rules in the grammar. The result of the grammar's work is a new word representing the sequence of the categories assigned to each subword of the input word. The segmentation of the input word and the result word of categories is called an analysis with respect to the grammar. For instance, let us represent the above regular expression as a grammar:

```
D -> ( ( 0, (1|2|3|4|5|6|7|8|9) ) | ( (1|2), (0|1|2|3|4|5|6|7|8|9) ) | (3,(0|1)) )
R -> .
M -> ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) )
Y -> (( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

Then for the date 10.11.2002 some of the possible analyses are:

```
 D  R    M  R      Y
10  .   11  .   2002
```

```
 D  R    D  R      Y
10  .   11  .   2002

 Y  R    Y  R      Y
10  .   11  .   2002

 D  R   Y   Y   R     Y
10  .   1   1   .   2002
```

where the categories are written above the corresponding subwords. As it is obvious from the example processing, this way of defining a grammar is not very useful because it produces too many wrong analyses. For a better processing with such kind of grammars one would like to be able to impose additional constraints over the analyses produced by the grammar. In the CLaRK System we made two extensions:

- cascaded application of regular grammar [Abney, 1996]; and
- extension of the rules with descriptions of the left and the right contexts of a subword recognized by the regular expression of the rule.

Cascaded regular grammar [Abney, 1996] is a sequence of regular grammars defined in such a way that the first grammar works over the input word and produces an output word of categories, the second grammar works over the output word of the first grammar, produces a new word of categories and so on. If one of the grammars in the sequence can produce more that one analysis, then these analyses are considered one by one in producing the analyses of the next grammar. The process of application of the cascaded regular grammar stops when all possibilities are finished. The output of the last grammar constitutes the analyses made by the cascaded regular grammar.

Example:

Let us take the previous grammar as a first grammar:

```
D -> ( ( 0, (1|2|3|4|5|6|7|8|9) ) | ( (1|2), (0|1|2|3|4|5|6|7|8|9) ) | (3,(0|1)) )
R -> .
M -> ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) )
Y -> (( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

And as a second grammar the following grammar with one rule:

```
Date -> (D,R,M,R,Y)
```

Applying the second grammar to the possible analyses of the first grammar, the only analysis that can be constructed is:

```
        Date
---------------------------
 D  R    M  R      Y
10  .   11  .   2002
```

and all other analyses from the first grammar will be ruled out as unacceptable.

One relaxation of the application scenario is that a given regular grammar in the sequence need not necessary recognise all letters in the input word. This means that the segmentation of the input word is such that the grammar recognises some of the subwords, but not all. An additional requirement suggested by [Abney, 1996] is the so-called *longest match*, which is a way to choose one of the possible

analyses for a grammar. The longest match strategy requires that the recognised subwords from left to right have the longest length possible. Thus the segmentation of the input word starts from left and tries to find the first longest subwords that can be recognised by the grammar and so on to the end of the word. The output word in this case consists of a sequence of unrecognised subwords of the input word and the categories of the recognised subwords in the order in which the subwords are found in the input word. In the CLaRK System we have implemented several more match modes: *shortest match*, *Any Up* and *Any Down* which allows the user to choose the most appropriate strategy for the task at hand. The shortest match requires that the recognized subwords must have the least possible length. The other two modes involve a kind of backtracking approaches in which decisions taken during recognition phase are reconsidered later. In the case of Any Up mode, the recognition of a subword starts with recognition of the shortest subword. In case of subsequent matching failure, the recognizer returns to this point and takes a little longer subword (if possible) and proceeds again with the rest of the input word. In the case of Any Down, the recognizer starts with the longest subword and reduces its length if needed.

The second extension of the regular grammars realised in the CLaRK System concerns the description of the left and the right context of a given subword recognised by the regular expression of a rule. Thus the rules in the grammar will have the format:

```
C -> LC : R : RC
```

where `C` is the category, `LC` is a regular expression describing the left context of the words recognisable by the rule, `R` is a regular expression describing the set of words recognisable by the rule, `RC` is a regular expression describing the right context of the words recognisable by the rule. The regular expressions `LC` and `RC` can be empty and then there are no constraints over the left and the right context. We envisage the use of such rules for tasks as sentence boundary recognition where one has to take a look at the word after the full stop in order to determine whether this full stop is marking the end of the sentence or not. The user can control the strategy for applying each of these rules by specifying longest or shortest match for them.

To demonstrate this kind of rules we rewrite the above grammar in the following way:

```
G1:
   M -> . : ( (0,(1|2|3|4|5|6|7|8|9)) | (1,(0|1|2)) ) : .
```

(the left context is a dot and the right context is a dot)

```
G2:
   D -> (( 0, (1|2|3|4|5|6|7|8|9) ) | ( (1|2), (0|1|2|3|4|5|6|7|8|9) ) | (3,(0|1))) : .
```

(the left context is empty)

```
   Y ->  . : (( (1|2|3|4|5|6|7|8|9), (0|1|2|3|4|5|6|7|8|9)*))
```

(the right context is empty)

```
G3:
   R ->  .
```

(both contexts are empty)

```
G4:
   Date -> (D,R,M,R,Y)
```

(both contexts are empty)

Longest match is required in all rules of this grammar although in some it is not obligatory.

The analysis will be in four steps:

```
Step1:

             M
    10   .   11   .   2002

Step2:

     D       M         Y
    10   .   11   .   2002

Step3:

     D  R    M  R      Y
    10   .   11   .   2002

Step4:

            Date
    ---------------------
     D  R    M  R     Y
    10   .   11   .   2002
```

In the next section we describe how cascaded regular grammars can be applied to XML documents.

2.2.2 Cascaded Regular Grammars in the CLaRK System

The basic the CLaRK mechanism for linguistic processing of text corpora is the regular grammar processor. The application of the regular grammars to XML documents is connected with the following problems:

- how to treat the XML document as an input word for a regular grammar;
- how should the returned grammar category be incorporated into the XML document; and
- what kind of 'letters' to be used in the regular expressions so that they correspond to the 'letters' in the XML document.

The solutions to these problems are described in the next paragraphs.

First of all, we accept that each grammar works on the content of an element in an XML document. The content of each XML element (excluding the EMPTY elements on which regular grammar cannot be applied) is either a sequence of XML elements, or text, or both.

When the content of the element to which the grammar will be applied is a text we have two choices:

1. we can accept that the 'letters' of the grammars are the codes of the symbols in the encoding of the text; or
2. we can segment the text in meaningful non-overlapping chunks (or in usual terminology tokens) and treat them as 'letters' of the grammars.

In the CLaRK System we have adopted the second approach. Each text content of an element is first tokenized by a tokenizer[4] and is then used as an input for grammars. Additionally, each token receives a unique type. For instance, the content of the following element

```
<s>John loves Mary who is in love with Peter</s>
```

can be segmented as follows:

```
"John"  CAPITALFIRSTWORD
" "     SPACE
"loves" WORD
" "     SPACE
"Mary"  CAPITALFIRSTWORD
" "     SPACE
"who"   WORD
" "     SPACE
"is"    WORD
" "     SPACE
"in"    WORD
" "     SPACE
"love"  WORD
" "     SPACE
"with"  WORD
" "     SPACE
"Peter" CAPITALFIRSTWORD
```

Here on each line in double quotes one token from the text followed by its token type is presented.

Therefore when a text is considered as an input word for a grammar it is represented as a sequence of tokens. How can we refer now to the tokens in the regular expressions in the grammars? The most simple way is by tokens. We decided to go further and to enlarge the means for describing tokens with the so called *token descriptions* which correspond to the letter descriptions in the above section on regular expressions. In the token descriptions we use strings (sequences of characters), wildcard symbols `%` for zero or more symbols, `#` for one or more symbols, `@` for zero or one symbol, and token categories. Each token description is matching exactly one token in the input word.

We divide the token descriptions into two types - those that are interpreted directly as tokens and others that are interpreted as token types first and then as tokens belonging to these token types.

The first kind of token descriptions is represented as a *string* enclosed in double quotes. The string is interpreted as one token with respect to the current tokenizer. If the string doesn't contain a wildcard symbol then it represents exactly one token. If the string contains the wildcard symbols `#` or `%` then it denotes an infinite set of tokens depending on the symbols that are replaced for the wildcard symbol. This is not a problem in the system because the token description is always matched by a token in the input word. In a similar way the other wildcard symbol is treated, but zero or one symbol is put in its place. One token description may contain more than one wildcard symbol.

Examples:

`"Peter"` as a token description could be matched only by the last token in the above example.

`"lov#"` could be matched by the tokens `"loves"` and `"love"`

`"lov@"` is matched only by `"love"`

`"@"` is matched by the token corresponding to the intervals `" "`

`"#h#"` is matched by `"John"` and to `"who"`

[4] There are built-in tokenizers which are always available and there is also a mechanism for defining new tokenizers by the user. Tokenizers are described below.

"#" is matched by any of the tokens including the spaces.

The wildcard symbols #, %, @ and the double quote symbols " can be used in the token expressions by using the symbol ^ as a escape symbol. Thus "^#" matches the token "#". The symbol ^ is represented as ^^.

The second kind of token descriptions is represented simply as a *string* preceded by the dollar sign ('$') and which is not enclosed in double quotes or angle brackets - < >. The string is either a token type or a set of token types if it contains wildcard symbols. Then the type of the token in the input word is matched by the token types denoted by the string. If the token type of the token in the text is denoted by the token description, then the token is matched to the token description.

Examples:

$WORD is matched to "loves", "who", "is", "in", "love", "with"

$CAP# is matched to "John", "Mary" and "Peter"

#$WORD is matched to "John", "loves", "Mary", "who", "is", "in", "love", "with", and "Peter"

$# is matched to any of the tokens including the spaces.

Variables in Token Descriptions

The variables are an extension of the token description language. Each variable describes some substring in a token when initialised for a first time, then matches to the same substring in the same token, or in some next tokens. Using this mechanism of initialization and value checking the variables can ensure equality between substrings in one or more tokens in the input word. In this way such phenomena as agreement or vocal harmony can be expressed in the grammar.

Each variable consists of the symbol & followed by a single Latin letter. The scope of a variable is the grammar. The variable can be used in the return mark-up and in this case the value of the variable is copied in the return mark-up. If the variable doesn't have a value then its name is copied to the return mark-up together with the & sign.

Each variable has positive and/or negative constraints over the possible values. They are given by lists of token descriptions (they can contain wildcard symbols, but not other variables). These constraints, in the process of compilation, are represented as a finite state automaton and the variable is substituted with it. In this way a variable can be evaluated only if the token contains a substring that can be recognized by the variable automaton.

Because the regular language that CLaRK System supports includes negation, the variables can occur in positive or negative contexts. A variable in a positive context always is traversed. Then it is initialized if it doesn't have a value or if the variable already has a value, this value is checked. In a negative context it can happen that a variable is not traversed. If it is traversed it again is initialized if it doesn't have a value or if the variable already has a value, this value is checked.

Defining the scope of the variables to be the whole grammar allows for potential undesirable interactions between different uses of the same variables after the compilation of the grammar. Each grammar is compiled into a Finite State Automaton (FSA). The compilation consists of determinisation and minimization of a FSA constructed from the regular expressions. During the compilation the automaton for each occurrence of a variable can be merged with the rest of the automaton of the whole grammar. During such merge two different occurrences of the same variable can be ordered in a way which is missing in the original grammar.

"a&Abcbcb" "acb&Ab&Ab" "adb&Ab&Ab" &A = "c"|"d"

Here are some examples: **"A&N&G", "Nc&N&G"**. These token descriptions can be used in a rule to ensure the agreement in number and gender between an adjective and a noun.

Now we turn to the case when the content of a given element is a sequence of elements. For instance the above sentence can be represented as:

```
<s>
```

```
<N>John</N> <V>loves</V> <N>Mary</N> <Pron>who</Pron> <V>is</V>
<P>in</P> <N>love</N> <P>with</P> <N>Peter</N> </s>
```

At first sight the natural choice for the input word is the sequences of the tags of the elements: `<N> <V> <N> <Pron> <V> <P> <N> <P> <N>`, but when the encoding of the grammatical features turns out to be more sophisticate as the following:

```
<s>
  <w g="N">John</w>
  <w g="V">loves</w>
  <w g="N">Mary</w>
  <w g="Pron">who</w>
  <w g="V">is</w>
  <w g="P">in</w>
  <w g="N">love</w>
  <w g="P">with</w>
  <w g="N">Peter</w>
</s>
```

then the sequence of tags is simply `<w> <w>` ..., which is not acceptable as an input word. In order to solve this problem we substitute each element with a sequence of values. This sequence is determined by an XPath expression that is evaluated taking the element node as the context node. The sequence defined by an XPath expression is called *element value* in the CLaRK System. Thus each element in the content of the element is replaced by a sequence of text elements. For the above example a possible element value for tag `w` could be: **attribute::g**. This XPath expression returns the value of the attribute `g` for each element with tag `w`. Therefore a grammar working on the content of the above sentence will receive as an input word the sequence `"N" "V" "N" "Pron" "V" "P" "N" "P" "N"`. Besides such text elements, by using of XPath expressions one can point to an arbitrary node in the document, so that the element value is determined differently. At the moment the nodes returned by an XPath expression are processed in the following way:

1. if the returned node is an element node, then the tag of the node is returned with the additional information that this is a tag;
2. if the returned node is an attribute node, then its value is tokenised by an appropriate tokenizer.
3. if the returned node is a text node, then the text is tokenised by an appropriate tokenizer.

Within the regular expressions we use the angle brackets in order to denote the tags. We can use wildcard symbols in the tag name. Thus

`<p>` is matched with a tag `p`;

`<@>` is matched with all tags with length one.

`<#>` is matched with all tags.

The last problem when applying grammars to XML documents is how we to incorporate the category assigned to a given rule. In general we can accept that the category has to be encoded as XML mark-up in the document and that this mark-up could be very different depending on the appropriate DTD we are using. For instance, if we have a simple tagger (example is based on [Abney, 1996]):

```
"the"|"a" -> Det
"telescope"|"garden"|"boy" -> N
"slow"|"quick"|"lazy" -> Adj
"walks"|"see"|"sees"|"saw" -> V
"above"|"with"|"in" -> Prep
```

Then one possibility for representing of the categories as XML mark-up is by tags around the recognised words:

```
the boy with the telescope
```

becomes

```
<Det>the</Det><N>boy</N><Prep>with</Prep><Det>the</Det><N>telescope</N>
```

This encoding is straightforward but not very convenient when the given wordform is homonymous like:

```
"move" -> V
"move" -> N
```

In order to avoid such cases we decided that the category for each rule in the CLaRK System is a custom mark-up that substitutes the recognised word. Since in most cases we would also like to save the recognised word, we use the variable \w for the recognised word. For instance, the above example will be:

```
"the"|"a" -> <Det>\w</Det>
"telescope"|"garden"|"boy" -> <N>\w</N>
"slow"|"quick"|"lazy" -> <Adj>\w</Adj>
"walks"|"see"|"sees"|"saw" -> <V>\w</V>
"above"|"with"|"in" -> <Prep>\w</Prep>
```

The mark-up defining the category can be as complex as necessary. The variable \w can be repeated as many times as necessary (it can also be omitted). For instance, for "move" the rule could be:

```
"move" ->   <w aa="V;N">\w</w>
```

2.3 Unicode for multilingual text representation (Tokenization)

The representation of the XML documents inside the system is based on UNICODE. This allows the representation of texts in different languages in a natural way. In XML each text nodes is considered as a sequence of letters. In order to treat these text elements XML considers the content of each text element as a whole string that is unacceptable for corpus processing where one usually requires to distinguish wordforms, punctuation and other tokens in the text. In order to cope with this problem the CLaRK System supports a user-defined hierarchy of tokenizers. At the very basic level the user can define a tokenizer in terms of a set of token types. In this basic tokenizer each token type is defined by a set of UNICODE symbols. Above this basic level tokenizers the user can define other tokenizers for which the token types are defined as regular expressions over the tokens of some other tokenizer, so called parent tokenizer. In the system tokenizers are used in different processing modules. For each tokenizer an alphabetical order over the token types is defined. This order is used for operations as comparing two tokens, sorting and similar. This order can easily be changed by the user.

Sometimes in different parts of one document the user will want to apply different tokenizers. For instance in a multilingual corpus the sentences in different languages will need to be tokenized by different tokenizers. In order to allow this functionality, the system allows for attaching tokenizers to the documents via the DTD of the document. To each DTD the user can attach a tokenizer which will be used for tokenization of all textual elements of the documents corresponding to the DTD. Additionally the user can overwrite the DTD tokenizer for some of the elements attaching to them other tokenizers.

2.4 Constraints

Several mechanisms for imposing constraints over XML documents are available. The constraints cannot be stated by the standard XML technology. The following types of constraints are implemented in the CLaRK: 1) regular expression constraints - additional constraints over the content of given elements based on a document context; 2) number restriction constraints - cardinality constraints over the content of a document; 3) value constraints - restriction of the possible content or parent of an element in a document based on a context. The constraints are used in two modes: checking the validity of a document regarding a set of constraints; supporting the linguist in his/her work during the building of a corpus. The first mode allows the creation of constraints for the validation of a corpus according to given requirements. The second mode helps the underlying strategy of minimisation of the human labour.

General syntax of the constraints in the CLaRK system is the following:

```
(Selector, Condition, Event, Action)
```

where the selector defines in which node(s) in the document the constraint is applicable; the condition defines the state of the document when the constraint is applied. The condition is stated as an XPath expression which is evaluated with respect to each node selected by the selector. If the evaluation of the condition gives an approving result then the constraints are applied; the event defines some conditions of the system when this constraint is checked for application. Such events can be: the selection of a menu item, the pressing of key shortcut, some editing command as inserting a child or a parent and similar; the action defines the way of the actual application of the constraint. .

At the moment the following constraints are implemented in the system:

Regular Expression Constraints

In this kind of constraints the action is defined as a regular expression which is evaluated over the content of each element selected by the selector. If the word formed by the content of the element can be recognized as belonging to the language of the regular expression then the constraint is evaluated as true. Otherwise it is evaluated as false and an appropriate message is given. The user can navigate on the valid/invalid nodes for the constraint. The content of each element is treated as the content of the element when a grammar is applied to it (see above).

These constraints are useful when the content of an element is text or MIXED and the user wants to impose some constraints over the content or when the DTD defines the content of some kind of elements as a disjunction and the different disjuncts are realised in different contexts which can be determined by a XPath expression. In this way the regular expression constraints can be used for imposing restrictions in addition to these in the DTD making them more specific on the basis of the surrounding context. This is very useful, for example, when someone is compiling a dictionary. Usually the DTD defines some very general content model for the elements, but in a concrete lexical entry a more specific model is realized. For example, let us assume that the `<Paradigm>` element in a lexical entry is defined as a disjunction of the following kind:

```
<!ELEMENT Paradigm ((sg,pl)|(pres,past,participle))>
```

The first part of the definition concerns the paradigm of nouns and the second - the paradigm of verbs. The information about the part of speech is presented in another element with tag `POS` and thus it is not available to the DTD validator. In order to support the consistency, we could write two constraints, which to express the dependency between the contents of the element `POS` and the element `Paradigm`:

```
Selector:   /descendant::Paradigm[preceding-sibling::POS/text()="N"]
Action: (sg,pl)

Selector:   /descendant::Paradigm[preceding-sibling::POS/text()="V"]
Action: (pres,past,participle)
```

The first constraint is satisfied by each `Paradigm` element, whose neighbour - `POS` element preceding it, has textual value N and thus the lexical entry is of a noun and the content of the element `Paradigm` is a sequence of one sg element and one pl element. Similarly for the second constraint.

Number Constraints

This kind of constraints are defined in terms of an XPath expression, which is evaluated to a list of nodes, and MIN and MAX values where MIN and MAX are natural numbers. The constraint is satisfied (evaluated as true) if the length of the list returned by the XPath expression is between MIN and MAX. The two border values can be either fixed (static) or dynamic (dependent on the context). In the latter case, the values are determined by XPath expressions which at application time are evaluated on the initially selected contexts. When a border value is not specified, the corresponding constraint does not set an upper/lower restriction.

This kind of constraints can be useful for checking equal number of nodes of different type within a given context. For instance, let us suppose that for a finite set of words of the language $a^n b^n$ it is necessary to construct an XML representation. Then we can define an element `<word>` in the following way:

```
<!ELEMENT word (a*,b*)>
```

By regular expressions, however, it is impossible to specify that the a^s are always the same number as the b^s. It becomes very simple with number restriction constraints:

```
Selector:   /descendant::word
Min:        count(child::a)
Max:        count(child::a)
Action: Min <= count(child::b) <= Max
```

and similar constraint for the number of a^s.

The number constraints are very appropriate for stating general constraints over the whole document. Such constraints include arbitrary complex XPath expressions in their predicate part and otherwise they always select the root note if the predicate is satisfied in the document and require that there is exactly one such element.

Value Constraints

These constraints determine the possible children or the parent of an element in a document. These constraints apply when the user enters a new child or a new parent of an element. In both cases a list of possible children or parents are determined by the DTD, but depending on the context in the document an additional reduction of these lists is possible. In case the only possible child of an element is a text then these constraints determine the possible text values for the element. Let us take as an example the following definitions in a DTD:

```
<!ELEMENT np ((np, pp) | ...) >
<!ELEMENT vp ((vp, pp) | ...) >
```

which in part define that a PP can be attached to a NP or a VP. Then let us take the partially marked-up sentence:

```
<s> <np>The man</np><v>saw</v><np>the boy</np><pp>in the
garden</pp> </s>
```

For the `PP` "in the garden" there are still two possibilities for a parent - a `NP` or a `VP`. But if the user enters a new information then "saw the boy" is a `VP` then for the `PP` "in the garden" there is only one possible parent - a `VP`. This information can be encoded in the system as a value constraint for the parent of `PP` elements. In future versions of the system we envisage such kind of constraints to be compiled from grammar represented in a grammar development environment.

In the next paragraphs we present constraints of type "Some Children". This kind of constraints deal with the content of some elements. They determine the existence of certain values within the content of these elements. A value can be a token or an XML mark-up and the actual value for an element can be determined by the context. Thus a constraint of this kind works in the following way: first it determines to which elements in the document it is applicable, then for each such element in turn it determines which values are allowed and checks whether in the content of the element some of these values are presented as a token or an XML mark-up. If there is such a value, then the constraint chooses the next element. If there is no such a value, then the constraint offers to the user a possibility to choose one of the allowed values for this element and the selected value is added to the content as a child in a certain position. Additionally, there is a mechanism for filtering of the appropriate values on the basis of the context of the element.

This kind of constraints is very useful for filling of the content of elements with predetermined set of values as, for example, in dictionary construction. Within BulTreeBank, we use these constraints for manual disambiguation of morpho-syntactic tags of wordforms in the text. For each wordform we encode the appropriate morpho-syntactic information from the dictionary as two elements: `<aa>` element which contains a list of morpho-syntactic tags for the wordform separated by a semicolon, and `<ta>` element which contains the actual morpho-syntactic tag for this use of the wordform. Obviously the value of `<ta>` element has to be among the values in the list presented in the element `<aa>` for the same wordform. "Some Children" constraints are very appropriate in this case. Using different conditions and filters on the values we implemented and used more than 50 constraints during the manual disambiguation of wordforms in the "golden standard" of the project. It is important to mention that when the context determines only one possible value for some word, it is added automatically to the content of `<ta>` element and thus the constraint becomes a rule.

Similar constraints are constraints of the type "Some Attributes" which work on the values of some attribute.

Here follows an example application of the value constraints of type "Some Children" which support the user in the process of work. The task in the example is a creation of a small dictionary, given a list of words. In the result structure, for each word the Part-of-Speech characteristic must be determined and a set of specific features depending on the part of speech. Thus, for each POS characteristic a different structure frame will be used. For simplicity here we will restrict the POS possibilities to three: article, noun and verb. The specific features are as follows: article - (type: definite/indefinite); noun - (gender: masc/fem/neu; number: singular/plural; case: acc/dat); verb - (tense: present/past; transitivity: trans/intrans).

In XML terms, the word structure definition is the following:

```
<!ELEMENT word (wf , pos , (gender , number , case | tense , transitivity | type))>
```

Here **wf** contains the wordform and **pos** contains the part of speech characteristic.

The supporting task of the value constraints here is to follow the user's activities and whenever s/he determines the POS characteristic of a word the corresponding structure frame is automatically created. In the next step, the user will be asked to supply values for the corresponding features by interacting with a convenient graphical interface.

The constraints here (all of type "Some Children") will be split into two sets. The first set contains constraints working in automatic insertion mode which on the basis of certain user decisions build the frames structures. The constraints in the second set require the user to select appropriate values for certain features.

The decision-requiring constraints are defined as:

```
target: //pos[not(child::text())]
                          // for all not specified POS characteristics ...
source: "article", "noun", "verb"
                          // select a possible value from the list
```

```
target: //type[not(child::text())]
                            // for all not specified type characteristics ...
source: "definite","indefinite"
                            // select a possible value from the list

target: //gender[not(child::text())]
                            // for all not specified gender characteristics ...
source: "masc","fem","neu"
                            // select a possible value from the list

target: //tense[not(child::text())]
                            // for all not specified tense characteristics ...
source: "singular","plural"
                            // select a possible value from the list
```

The automatic decision making constraints are defined as:

```
target: //word[child::pos[text() = "article"]
                            // all words having POS article ...
<type/>
                            // there must be an element \markup{type} inserted

target: //word[child::pos[text() = "noun"]
                            // all words having POS noun ...
<gender/>
                            // there must be an element \markup{gender} inserted

target: //word[child::pos[text() = "noun"]
                            // all words having POS noun ...
<number/>
                            // there must be an element \markup{number} inserted
```

Note: Several more analogous constraint definitions must be created to cover all the cases.

Once defined, the constraints are put in a group and the whole group is applied. The definition of the constraints makes the application of certain constraints dependent on the application of others. Only the constraints for which the preconditions for application are available are executed. All the rest are standing in the background and waiting for their turn.

Having started the group application, the selection stops on the first word and the user is asked to specify the POS. Having done that the appropriate frame is automatically constructed and the selection is moved to the first unresolved feature in it. Having resolved all features within the current frame for which there is no automatic insertion rule, the selection is moved to the next word and the system expects again user decision and so on. Thus, the whole process of application consists of two types of phases: a user decision phase and a propagation phase of the consequences of the user decisions made on previous phases. In this way, there is no user interaction as long as non-trivial decisions have to be taken.

3 The CLaRK System

At the heart of the CLaRK System is the XML technology as a set of utilities for structuring, manipulation, management and visualization of data. We started with basic facilities for creation, editing, storing and querying of XML documents and developed further this inventory towards a powerful system for processing not only of single XML documents but of an integrated set of documents and constraints over them. The main goal of this development is to allow the user to add to the XML documents a desirable

semantics reflecting the user's goals. Inside the system, the core structure of the representation of the XML documents follows the DOM Level1 specification - see [DOM, 1998]. When an XML document is imported (or created) in the system it is stored in this internal representation and in this way the user has access to it only via the facilities of the system. This restriction allows us to support the consistency of the data represented in the system. We plan to exploit this feature of the system even further in future for automatic support of construction of XML documents that reflect the content of a corpus or a set of documents.

The CLaRK System includes the following components: *XML Engine*, *XML Editor*, *Internal Documents Database*, *Document Transformation*, *Tokenizer*, *Constraints Engine*, *XPath Engine*, *Regular Grammar Engine*, *MultiTool Processor*. In this section we describe each of these components in turn. Some of these components are not directly accessible by the user and they are used in the other components to support the corresponding functionality. The most important components of this type are the *XPath Engine* and the *Regular Grammar Engine*. The first is a module which evaluates XPath expressions over a document and the second is a module dealing with compilation of regular expressions into finite-state automata, determinization and minimization of the compiled automata.

3.1 XML Engine

XML Engine offers a full set of facilities for processing XML documents. This includes:

DTD compiler

The DTD compiler which compiles the elements, entities and attribute definitions in a DTD and represents them in an internal format. For the elements the internal format is a set of finite-state automata corresponding to the content definition of the elements. These automata are determined and minimized during the compilation. Attributes and entities are stored as hash-tables. Additionally in the internal format the user can write comments on the element, attribute and entity declarations.

Functions connected with the DTD compiler include creating, modifying and removing DTDs from the system. Also visualization adjustments are related with the DTD management: `DTD Text` and `Tree Layout`.

XML parser

This parser transforms an XML document into internal for the system DOM representation. During the parsing process the parser checks the well-formedness of the document and reports the corresponding errors. The parser is aware of different input/output file encodings: `UTF-8`, `UTF-16`, and about 34 ASCII encodings.

The third component is the *Validator*. This module checks the validity of the document with respect to a DTD. Each document which is loaded in the system has to be attached to a DTD. Once a document is parsed to the internal representation of the system, it can be saved in this internal representation and the next time when it is used it will not be necessary to be parsed again. The validator is active for the currently loaded document in the editor and when the user modifies the document the validator reports the changes in the validity of the document with pointers to the corresponding wrong elements.

3.2 Internal Documents Database

The *Internal Documents Database* is an essential part of the CLaRK system. It is the place where all documents saved in the system are stored. The main reason for maintaining such a database is that the system can work only with well-formed XML documents and having them stored in an internal format prevents the system from malfunctioning because of incorrect data. This database protects the documents from being corrupted. In addition, keeping the documents "invisible" for other applications allows saving additional system specific information with them. The interface of the documents with external applications can be done by import/export operations. A successfully imported document in the system has a guaranteed well-formed structure. An exported document from the system is always a

well-formed XML document stored in a file, which can be used by other XML oriented applications.

The Internal Document Database represents a set of repositories, called `Corpora` which contain the saved documents in the system. The different corpora are identified by unique (user supplied) names. Each corpus contains a set of documents which are stored independently from the documents of other corpora. No documents can be saved outside a corpus. The documents within a corpus are identified by unique names, i.e. no two documents can have the same name. If it is necessary that two or more documents should have the same name, they must be stored in different corpora.

The possible operations on corpora are:

1. *Creating a new corpus.* The user supplies a new corpus name which hasn't been used yet for another corpus.
2. *Removing a corpus.* The user selects a corpus to be removed. All documents contained in the selected corpus are removed as well. The removal is preceded by a confirmation warning message. There is one system protected corpus ('Root') which can not be removed. It is always presented in the system and it is used as a default corpus.
3. *Renaming a corpus.* The user can rename an existing corpus by supplying a new unique name. The 'Root' corpus is protected from renaming.

Corpora are document containers only, so it is not possible a corpus to contain another corpus. Each corpus has its own internal logical structure which facilitates the maintenance of many documents at the same time. The documents within a corpus are organized in *groups*. Each group can contain a set of documents, as well as a set of subgroups. Each group is identified by an unique name within the group in which it is included, but not necessary unique for the containing corpus. The supported operations on groups are: creating a new group or a subgroup of a group; renaming a group and removing a group. The user can include or exclude documents in a group.

The groups are logical unions of documents which do not impact physically the documents themselves. This means that if a document is removed (excluded) from a group, the document itself is not deleted from the database, but just excluded from this union. Thus the same document can be included in more than one group, as well as a document can be included in none.

Note: technically said, the groups contain not documents, but references to documents. In this way, if a document is presented in more than one group, the document itself is not multiplied for each containing group, but the single instance is referred by links from each group entry.

Observing the documents in the Internal Documents Database can be done in three views:

1. *by logical tree structure* - the hierarchical structure is visualized in a way that on the topmost level the set of available corpora is shown and under each of them the groups tree structure is visualized. Selecting a point in the hierarchy causes a visualization of its content in a separate window.
2. *by all documents in a corpus* - all documents which are presented in a certain corpus are represented in a table, showing the document names and some other information. The user can switch to another corpus in order to see its content in a table. In this view the groups information can not be observed.
3. *by all documents in the whole database* - all documents from all corpora are represented in one table. The information shown in the table is the same as the one from the previous view, plus a source corpus name for each document.

3.3 XML Editor

The access to the system is via a structure-driven editor which allows the user to edit and manipulate XML documents. Each loaded into the editor document is presented to the user in two or more views.

One of these views reflects the tree structure of the document as described in the previous section. The other views of the document are textual. Each textual view shows the tags and the text content of the document. The tags in the textual view are separate elements from the rest of the text and can not be edited. The user has the possibility to attach to each textual view a filter (Text layout) which determines the tags and the content of which elements to be displayed in the view. This option allows the user to hide some of the information in the document and to concentrate on the rest of the information. With different textual views of the same document the user can attach different layouts. Here, there are different tags alignment adjustments as well. An advanced option here is that different tags and text content can be colorized on the basis of the context they appear. These color settings (called `Color Schemes`, rely on XPath expression predicates. If the evaluation of such a predicate on a node returns an approving result, the corresponding node is colorized in a certain color. Otherwise, the default color for the node is used. Each XPath predicate is wrapped in a rule in which the target color is specified. Each *Color Scheme* can contain a set of rules, but only one can be applied to a node (the first one which matches).

The tree view panel also supports color layout adjustments. The tree structure can not be changed, but the nodes' labels can be controlled by the user. Instead of seeing the tag names, a template can be defined to show some information local for the specific node (for example, attributes names/values). The template relies on XPath to collect the data to be shown in the node's label. Thus any information relative to the specific node can be shown. The template itself can be a mixture of XPath expressions and static text, which appears unchanged. Additionally, the color of the text label can be adjusted.

The editor supports a full set of editing operations as copy, cut, paste and so on. These operations are consistent with the XML structure of a document. Thus the user can copy or delete a whole subtree of the document. Some of these operations as search and replace are defined in terms of XPath expressions. This allows the user to search not only in textual content of the document but also with respect to the XML mark-up. The most powerful operation here is the XPath Transformations tool. This operation is used for various commands for restructuring the document. Generally, the scenario is the following: (1) a list of nodes (subtrees, text elements) is chosen by the Source XPath expression. In this way the elements which will be copied or moved in the document are defined; (2) a list of nodes is chosen by the Target XPath expression. In this way the place(s) where the source elements will be copied or moved are defined; (3) the elements from source list are attached to the elements of the target list. There are several options defining the way of performing of the above action. These concern such things as whether the elements of the source are copied or cut from the document before they are attached to the target, the mapping between the source and target elements - it is possible for the source elements to be attached to each element of the target, or each element of the source to be attached to the corresponding element of the target. Via different options this operation becomes very powerful means for document modification or entering new information in case the source is given not as an XPath expression but as a fragment of XML document or text. It is also possible the source expressions to be evaluated on different external documents and thus allowing their merging.

The editor allows editing of the document textual content or its structure. The editing of the structure is supported by the attached to the document DTD. When the cursor is located at some point in the document structure, the user can insert an attribute, a child, a sibling or a parent of the pointed element. In all cases the DTD is consulted and the list of the allowed for this position tags/values is offered to the user.

In the system there are several global graphical interface options which can be controlled. There is a standard font chooser which controls the fonts of the different components independently. A visuals manager controls the default coloring of different elements in the editor: text and tree views' backgrounds; attribute values, tags, text and comments colors. The style in which graphical components are drawn (i.e. the *Look and Feel*) can be tuned for easier new user environment adaptation.

The system supports different user defined keyboard layouts. This option can be used for languages which use not standard letters or letters which do not appear on the available keyboard. Other usages are in cases when the available auxiliary system layouts are not convenient in a way or there is no support at all. In such cases the user can define custom key-to-character bindings. The definitions are done with

the help of graphical user interface components. Once defined, each layout can be saved and used or modified later. The system can work with several layouts at the same time (if it is needed). The layouts are defined by unique names.

3.4 Graphical Tree View

The Graphical Tree View is a means for drawing of tree structures encoded in XML. The result structure is not necessary to follow the XML logical structure but it can be user defined. The structure definition is based on XPath expressions. The view of the result drawing follows a defined in advance layout.

The Graphical Tree Layout is a means for drawing arbitrary tree structures represented in XML. The resulting graphical representation obeys different user adjustments, like colors and shapes rendering, text and structure definition and filtering and others.

The main graphical objects which can be used for nodes representation are: rectangles, rounded rectangles and ellipses. The user can specify their outline color and thickness, background color, text label inside (font, color and content). The nodes in the drawing are connected with lines, the appearance of which is again user defined: color and thickness. Additionally, there are cross-branches links available. They can connect any nodes in the drawing with arcs for which the curvature can be adjusted (to avoid overlapping with other lines and arcs) .

The layout itself represents a set of rules, each of which corresponding to one shape definition. Each rule has a conditional part which defines to what kind of nodes the rule is applicable (element, text or comment nodes or nodes, appearing in certain XPath defined context). If a condition for a rule is satisfied for a certain node, a new graphical object appears on the drawing canvas. The appearance of the object is defined in the rule.

Another important part of each rule is the Children Definition section which determines the nodes whose graphical representations will appear as children of the graphical representation of the current node. The children definition is based on XPath expression and this allows visualization of nodes which are not direct children of the current node (or even nodes which do not belong at all to the current structure) as child nodes. The default value of the children definition for each rule is child::*, i.e. all direct child nodes.

If none of the rules in a layout are applicable for a certain node, there are three default rules, one of which always succeeds depending on the node type (element, text or comment).

An important section in each layout is the Structure roots definition section. It defines which nodes in the current document are suitable to be roots of a structure to be visualized. It contains an XPath expression which is used as a condition and if it is evaluated successfully for a certain node, the graphical representation building starts from it. Otherwise, the system searches for the closest ancestor which satisfies the condition.

In order to visualize a document (or a part of a document), it must be opened in the system editor. Having selected a node in the document, the menu item View/Graphical Tree View must be chosen which shows the graphical representation in a separate window.

An example usage of this kind of views is related with ontology management. Assuming that an ontology is encoded in RDF format, one inconvenience could be that the definitions of the concepts are in a way linear in the document. I.e. the definitions follow one after another and the relationships between them are not presented in the logical structure of the XML document. Thus, if no specialized tool is used the observation of the hierarchical structure is very hard (sometimes impossible). Solving this problem here is relatively simple by defining a suitable graphical layout. The key point is in the definition of the child nodes for each node. Instead of showing the content (which is the default behaviour) of a node representing a concept, we will collect all nodes (representatives of concepts) which have a *sub-class-of* relation referring to this node (concept). In this way, the hierarchical structure will be shown naturally as a tree. Another benefit of this view is that it is synchronized with the original document and any selection on it moves the selection in the source document to the corresponding original position. This

helps in cases when a modification on the source document is needed.

3.5 Cross Document Synchronization

This feature allows several documents opened in the system editor to be navigated in parallel. This is convenient for navigation in aligned text stored in separate documents. Another application is when in a document there are references to other documents and the referred elements has to be found and shown. There are two mechanisms for synchronization which use different approaches.

3.5.1 Absolute Path Synchronization

This function gives the facility two or more documents with identical structures to be navigated in parallel in the editor. There is no restriction how many documents to be synchronized at the same time. The synchronized documents form a group in which, if the selection in one document is changed this causes a selection change in the rest group members. In this aspect, the synchronization is symmetric, i.e it does not matter which document is current and which one is synchronized.

This feature is useful when documents have to be compared manually for some reason. Another application is when there is a set of documents representing parallel aligned data (in example, parallel corpora where each language is in a separate document).

The synchronization, i.e. the distribution of the selection from one document to the others is done on the basis of the tree path location of the initially selected node. When a selection event is performed to a certain node, its path to the tree root is calculated, observing for each node in the path its preceding siblings' info. The calculation of this tree path results in an XPath expression which deterministically points to the selected node. In the next step, in all the rest synchronization group members this XPath expression is evaluated and if it returns a node, the corresponding document selection is moved to it. Otherwise, the document's previous selection is cleared. Thus if two documents are structurally identical, for each node in one of them, there will be a corresponding node in the other one. If having selected a node in a document, the selection disappears in the other document(s), it means that this node is in a way unique for this document.

3.5.2 Rule-based Synchronization

The Synchronization Rules is a means for establishing connections between opened documents in the system editor. The connections are expressed as distributions of a selection in one document to selections in other documents. The connections are based on XPath expressions processing and evaluation.

When a connection between the current document and another referent document is established each change of the selection in the current one is registered. If the new selection satisfies certain conditions, the connection is activated and a new XPath expression is generated on the basis of a pattern. The new XPath is evaluated in the referent document and the result is selected (in case it is not an empty node-set).

For establishing a connection of this type, exactly two documents are required: a current document in which the user works and a referent document in which the selection moves depending on the Sync rule parameters and the selection in the current document. This type of connections is asymmetric. There is no restriction on the number of connections which can take a certain document as a current one. In this respect, the user can connect one document with several referent documents and in this way navigating in one document causes distribution of selections in different documents in the same time.

An example usage of this facility is when a certain document is explored and simultaneous look-ups in a dictionary document are needed. In this case a connection between the observed and the dictionary documents can be established. Whenever a certain word/expression to be looked up is selected, the system extracts the necessary local information and performs a search in the dictionary document. If

the searched entry is found it is selected in the background target document. Thus moving from word to word in the observed document leads to automatic showing the corresponding entries from the dictionary.

Another application of this tool is when different documents are connected in some way by references. Whenever a reference in one document is selected, the referent entity in the corresponding document is selected.

3.6 Additional Tools and Facilities

Besides the fundamental tools presented so far, there is a great number of additional facilities for supporting corpus development implemented in the CLaRK system:

XPath Extractor

The Extractor is a tool for extracting elements with or without their contents from XML documents in a new document. For instance this can be used to extract all sentences in certain documents that meet some condition. The condition is stated as an XPath expression.

XPath Remove tool

The Remove tool allows the user to delete some elements with or without their contents that meet a certain condition stated again as an XPath expression.

XPath Rename tool

This tool allows the user to rename `Element` nodes in a document selected by an XPath expression. A new tagname is expected to be specified. The selected nodes are renamed without changing their attributes and content.

XPath Insert ... tool

This set of tools allows for XPath controlled structured data insertion. The following paragraphs describe them in details.

XPath Insert Attribute tool

This tool gives the possibility to set certain attributes to nodes selected by an XPath expression. The user specifies an attribute name and value. Additionally, s/he can tune the tool to set attributes only to nodes which do not have such yet.

XPath Insert Child tool

This tool allows the user to insert certain child nodes in the content of `Element` nodes. The target nodes which the insertion will be applied to are selected by an XPath expression. The result from the evaluation of the XPath expression must be a list of `Element` nodes. All the other types of nodes are discarded. This tool can insert two types of child nodes: `Element` and `Text` nodes. If the new children are of type `Element`, the tool expects from the user to supply a valid tag name. Otherwise, i.e. when the new children are `Text` nodes, the tool accepts any non-empty textual data. The user can also set on which position the new children will appear in their parents' content. Here the counting starts from 0, i.e. the first child is denoted by 0, the second - by 1, etc.

XPath Insert Parent tool

This tool enables insertion of parent `Element` nodes of selected by an XPath expression nodes. The selected target nodes by the XPath expression can be either `Element` or `Text` nodes. Any other types of nodes are discarded from the selection. This tool expects from the user to specify a valid tag name for the new parent nodes.

XPath Insert Sibling tool

This tool allows inserting sibling nodes of nodes selected by an XPath expression. The target selected nodes can be of any type, but `Attribute` nodes. If the root node is selected, it is discarded during the processing time. The new nodes for insertion can be of type `Element` or `Text`. If the new nodes are of type `Element`, the tool expects from the user to supply a valid tag name. Otherwise, i.e. when the new

siblings are `Text` nodes, the tool accepts any non-empty textual data. The user can also set the position where the new sibling nodes will appear. The options are: **previous** (preceding the target node sibling) and **next** (following the target node sibling).

Statistics tool

The Statistics tool is used for counting the number of nodes or/and token occurrences in XML document(s). The items to be counted initially are selected by an XPath expression. The selection returned by the XPath evaluation is a node set. For the nodes from the selection the user defines *Value Keys*. Each key contains an XPath expression which is meant to point the essential properties of the selection. For each node of the initial selection the values from the *Value Keys* are calculated independently. If for two nodes the corresponding calculated values are the same, they are assumed to belong to the same class. In this way each of the selected nodes is classified in a set. If the statistics has to be applied not only on XML nodes, but on tokens the user must select a tokenizer. In this way the text nodes will be segmented into meaningful tokens. In addition the user can filter the tokens by category in order to receive information only for certain types of tokens. If no tokenizer is selected, the text nodes will be processed as a whole node.

The result from the statistics application is a list of all classes formed by the selected nodes. The information which is kept for each class is:

Searched Item - the item found by the selection (tag name or token);

Item Category - the category of the search item, if the item is a token;

Number of occurrences - the number of items from the selection which belong to the class;

Percentage - the percentage of the items belonging to the class, compared to the rest from the selection;

Keys Value - a string representation of the value(s) for the class.

The result is visualized in a table and can be stored in an XML document.

Sort

The Sorting tool is concerned with sorting elements of a document according to some keys defined over these elements. The sorting is defined in terms of two XPath expressions. The first expression determines which elements will be sorted. This expression is evaluated with respect to the root of the document as a context node. The second XPath expression defines the key for each element and it is evaluated for each node returned by the first XPath expression. The list of nodes returned by the first expression is sorted according to the keys of the nodes. Afterwards the nodes are returned in the document in the new order.

Concordancer

A concordance tool is implemented on the basis of the XPath engine, regular grammar engine and a sorting module. The concordance tool is useful for searching of some kind of units within some bigger units. For instance, a word within a sentence, a phrase within a paragraph and similar. The bigger element is called here a *context* and the smaller element is called *item*. The context is defined by an XPath expression. The item can be defined by an XPath expression or by a regular grammar. There are additional possibilities to restrict the context by regular grammars or XPath expressions. The found units are stored in a new document and presented to the user in a table format. The user can also open the document as an ordinary XML document and use all the tools available in the system in order to process further this document.

For example, in case of appropriately marked-up corpus one can extract all verbs and order them with respect to the first noun on the right hand side of the verb (not necessarily the first word on the right hand side).

4 Tree Intersection Tool

This tool represents a means for comparing tree structures. It calculates the maximal tree structure which is presented in each of the input structures. The result structure (the intersection) is opened in the system editor. The differences and the places where they appear are also preserved in internal structures. On a later stage this information is used to support the user to find the correct target structure by expanding manually the intersection. When an intersection document is opened, the system switches to a special performance mode and some of the tree restructuring functions are disabled. Also some new tool specific functions appear. The structure difference data turns into constraints and the user expands the intersection by choosing pop-up menu items instead of manual typing. Every time the user takes a decision by choosing an item, this information is spread on the whole result document and the intersection is extended. The expansion directions for a selected (unresolved) position can be: specifying an attribute name or value; specifying content nodes, sibling nodes or parent nodes. Along with confirming expansion direction entries, the user also can reject certain entries (a kind of negation) and thus expanding the intersection structure.

In order to compare (intersect) tree structures, the user has to point out the source structures. They can be located in one document or in more than one. The topmost node of each structure to be processed within a document is pointed by an XPath expression. If each document contains a single structure the XPath expression selects the document's root (i.e. /self::*).

Tree Comparing Algorithm

The construction of an intersection is incremental. It starts from the root of each structure and tries to find as many common nodes as possible, moving down to all input structures. When a common node for all structures is found, it is inserted in the result structure. The algorithm combines a top-down with a bottom-up matching strategy, i.e. the construction of the intersection starts from the root to the leaves and when it is no longer possible to find nodes in common which are directly assigned to the structure the system tries to find common nodes on lower levels discarding some nodes on the paths to the roots. When such common nodes are detected they are attached to the first common grand parents from the result structure (at worst, to the root).

There are different criteria for finding common nodes. The main factor in matching is the node names for elements; the string content for text and comment nodes and names and values for attributes. The contexts of the nodes in the structures (parents, children, siblings) are also very important. The nodes matching module is aware of the `Element Features` defined in the corresponding DTDs. This is crucial when within a structure there are element nodes with equal names, children of the same parents. Then a distinction is needed to prevent incorrect structures matching. With the help of well defined `Element Values`, the processor is enabled to 'see ahead' and to make the correct matching.

5 Different Processing Tools Integration

In order to make the processing in CLaRK more flexible, most of the system tools are designed in a certain unified scheme. Each tool's settings needed for application are stored in XML document (called *Tool Query*). On the base of such descriptions, no other things are needed for tools applications. Once a tool query is created, it can be used many times for applying on different data. These queries can also contain information about the input and the output (result) data for a certain application.

Each tool from this group supports two modes of application: on the currently opened document in the system editor and on document(s) from the internal documents database. In the latter case, the processing efficiency is better and it is recommended for working with bigger amounts of data.

This architecture allows integrating different tool applications in a complete processing procedures. The integrated procedure definitions consist of lists of `Tool Queries`. They are applied successively along the lists. The result from a query in a list is an input for the next query in the list. The last query produces the final result.

The tool which supports this multiple tool integration is called `MultiQuery Tool`. It is also designed in this integration architecture, so MultiTool queries can be imbedded in other MultiTool queries. Thus, more complex tasks can be decomposed in more simple ones.

5.1 Control Operators

The `Control` operators allow changing the order of application of the queries in the MultiQuery tool. The usual order of applications starts from the first one and proceeds one by one up to the last one. Using Control operators some queries can be applied only if certain conditions are true. Such conditions are: the true or false value of a result from an XPath evaluation; whether the preceding single tool application has or has not modified the working document; or unconditional (always succeeding). When a condition for a Control is true, the next query (or another Control), which will be applied, is defined in the control itself. Otherwise, the application proceeds with the next entry in the list. The Control operators address their targets (queries or controls to be applied in case of success) by pointing their labels. Each entry in the list of the queries can have a label (unique identifier) which can be referred by to control operators. It is an error if a Control operator uses a target label which does not exist.

By using the Controls-labels technique, the user can model the famous *IF-THEN-ELSE* and *WHILE-CONDITION-DO* structures in order to make the processing more flexible. The composition of different Controls allows the user to create varied 'programs' or 'scripts' capable of doing certain jobs. It is up to the user to create efficient and reliable processing procedures.

5.2 Conditions

The `Conditions` are operators which perform certain checks on the current working document. The conditions can cause reconsidering the decisions taken so far and producing new result documents. Different decisions for a certain document can be taken by the *Grammar* tool and the *(Value) Constraints* tool. The condition operators can be used ONLY in Multiple Apply mode of the tool. The usage on the current document is not allowed because of efficiency reasons (multiple backtracking events can cause the system to work very slowly).

When a condition check fails, it causes the system to reconsider the latest decision, taken on a place of a choice point and recovering the working document to the state when the previous decision was taken. If a new decision can be taken, the system proceeds the application with it. Otherwise, the system continues searching backwards for another choice point where a new decision can be taken. If no solution for a condition is found, the system terminates the current application.

There are two types of Conditions which can be used:

`XPath based` - an XPath expression is evaluated on the working document and if the result is approving (not empty node-set, not empty string, positive number or true boolean value) the condition is satisfied;

`Value Constraint based` - the condition specifies a (Value) Constraints query which contains constraints to be applied in validation mode on the working document. If one of the constraints is not satisfied, the whole condition fails.

6 Future development

The CLaRK system has been used very intensively within the BulTreeBank Project at the Linguistic Modelling Laboratory - see [Simov, Popova and Osenova, 2001] and [Simov et al., 2002]. We plan to extend the system in the following directions:

External programs. A mechanism for calling external programs which receive as input fragments of an XML document and returns also fragments of XML document. We envisage the actual communication to the external programs to be implemented via transformations of fragments of documents to and from

special interface XML documents. In this way an external program will be declared within the system only once and the user will be able to use the program with XML documents with different structure.

Schemes of dependencies between elements in several documents. This is in connection with databases. We can consider each XML DTD as a conceptual scheme over XML documents. Then we can use a set of DTDs to describe interconnected schemes. We plan to implement support for such schemes. Because the task can prove to be very hard we will start with one basic DTD and auxiliary DTDs defining interconnections in table format.

We plan to extend the set of events and actions available to the user for defining the constraints. The target here will be a macro language for definitions of actions. Also we plan to make the constraints more active and as an activating event will be used the result from evaluation of some other constraint. In this way we will have mechanisms for propagation of information from one constraint to others.

6.1 Acknowledgements

We would like to thank to Atanas Kiryakov and Marin Dimitrov from OntoText Lab., Sirma AI, Sofia (http://www.OntoText.com) for their participation in the initial stages of the development of the CLaRK System.

Also we would like to Zdravko Peev for his great work during early stages of the implementation of the system.

We would like to thank to our colleagues from BulTreeBank Project: Petya Osenova, Milena Slavcheva, Sia Kolkovska, Elisaveta Balabanova and Dimitar Doikoff for their patience during the use of the system in their work.

We would like to thank Petya Osenova and Gergana Popova for their help with the English.

We would like to thank Tylman Ule for his reading of the documentation and pointing us to some inconsistencies.

We would like to thank all our users for their enthusiasm in using the system and for all the errors they found in it.

All errors in the text are ours of course.

References

[Abney, 1991] Steve Abney. 1991. *Parsing By Chunks.* In: *Robert Berwick, Steven Abney and Carol Tenny (eds.), Principle-Based Parsing.* Kluwer Academic Publishers, Dordrecht.

[Abney, 1996] Steve Abney. 1996. *Partial Parsing via Finite-State Cascades.* In: *Proceedings of the ESSLLI'96 Robust Parsing Workshop.* Prague, Czech Republic.

[XCES, 2001] Corpus Encoding Standard. 2001. *XCES: Corpus Encoding Standard for XML.* Vassar College, New York, USA. http://www.cs.vassar.edu/XCES/

[DOM, 1998] DOM. 1998. *Document Object Model (DOM) Level 1. Specification Version 1.0.* W3C Recommendation. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001

[Simov, Popova and Osenova, 2001] K. Simov, G. Popova, P. Osenova. 2001. *HPSG-based syntactic treebank of Bulgarian (BulTreeBank).* In: *Proceedings of Corpus linguistics 2001,* Lancaster, UK.

[Simov et al., 2002] K.Simov, P.Osenova, M.Slavcheva, S.Kolkovska, E.Balabanova, D.Doikoff, K.Ivanova, A.Simov, M.Kouylekov. 2002. *Building a Linguistically Interpreted Corpus of Bulgarian: the BulTreeBank.* In: Proceedings from the LREC conference, Canary Islands, Spain.

[TEI, 2001] Text Encoding Initiative. 1997. *Guidelines for Electronic Text Encoding and Interchange.* Sperberg-McQueen C.M., Burnard L (eds).

[XML, 2000] XML. 2000. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/REC-xml

[XPath, 1999] XPath. 1999. *XML Path Language (XPath) version 1.0.* W3C Recommendation. http://www.w3.org/TR/xpath

[XSLT, 1999] XSLT. 1999. *XSL Transformations (XSLT). version 1.0.* W3C Recommendation. http://www.w3.org/TR/xslt