Testing structural properties in textual data: beyond document grammars

Introduction

This article describes research carried out in the project "Secondary information structuring and comparative discourse analysis" (SEKIMO), which is part of the research group "Text-technological modeling of information" and is funded by the German Research Council (DFG). In our project, we use XML document grammars, i.e. DTDs (Bray et al., 2000), XML Schema (Thompson et al., 2001) and Relax NG (Clark and Murata, 2001) to formalize and interrelate linguistic phenomena in typologically diverse languages. The document grammars differ in what they describe, that is morphosyntactic structures, semantic relations and discourse functions, and in the granularity of the description; i.e. there are language or dialogue type specific document grammars on the one hand and document grammars of a more general kind on the other hand. At the level of secondary information structuring, we interrelate the document grammars, sometimes creating 'intermediate' document grammars in order to connect the specific and general levels of linguistic description. All document grammars are developed on the basis of and applied to dialogue and text corpora in different languages. (For more information about the project, see www.text-technology.de).

Schema languages usually define grammatical constraints on document structures, i.e. hierarchical relations between elements in a tree-like structure. Especially but not only for the linguistic phenomena we want to describe, it seems useful to complement the concept of hierarchical validation with a methodology for defining and applying other structural constraints as there are several limitations in implementing appropriate document grammars. The main benefits of this methodology are:

- Addition of constraints
  which are hard to express using schema languages
- Independent formulation of constraints;
  adding new constraints does not require changes to document schema
- Classification of information items;
  assigning classes based on fulfillment of constraints.

We will exemplify this in reference to the document in Fig. 1, which is based on the English part of the MULTEXT-EAST corpus (Ide and Véronis, 1998).

```
<corpus>
  <p>
    <s>
      <name>Ministry of Truth</name>
      , -
      <name>Minitrue</name>
      , in
      <name>Newspeak</name>
      - was startlingly different from any other object in sight.
    </s>
    <s>It was an enormous pyramidal structure of glittering white concrete, soaring up, terrace
      after terrace, 300 metres into the air.</s>
    <s>
      From where
      <name>Winston</name>
      stood it was just possible to read, picked out on its white face in elegant lettering, the
      three slogans of the
      <name>Party</name>
      :
      <q>War is peace</q>
      <q>Freedom is slavery</q>
      <q>Ignorance is strength.</q>
    </s>
  </p>
</corpus>
```

Fig. 1 Annotation of a paragraph from “1984”

Hierarchical constraints for the name element are for example that it has to occur inside a sentence, here tagged as s. These constraints can also be described in terms of contextual constraints for name, i.e. its ancestor has to be an s element. The hierarchical and the contextual constraints are visualized in Fig. 2.

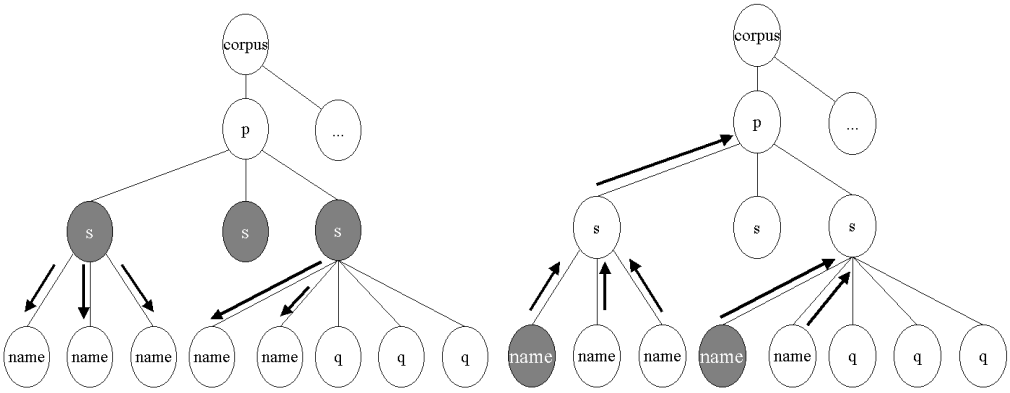

Fig 2. Hierarchical and contextual constraints on elements

In the left-hand part of Fig. 2 there is the hierarchical constraint for name elements, i.e. they occur in the content model of sentences s. In the right-hand part the relation between name and s is described as a contextual feature of name, visualized with arrows pointing from name elements to s elements. This feature is shared by all name elements. Other features are shared only by some name elements. For example, the first occurrence of name is at the beginning of a sentence s. The same is true for the fourth occurrence of name, visualized by the grey background of the two elements. They can be further classified: the first name is inside a sentence s which is at the beginning of a paragraph p, the fourth name is inside a subsequent sentence. In other words, the contextual features of elements can be organized in terms of a class structure, with classes containing general properties and subclasses, which define more specific properties respectively.

For tasks like visualizing, modeling, querying and checking consistency of text, it might be very useful to describe the contextual features of elements and arranging them in a

class structure. A document containing such descriptions we call "context specification document" (CSD). In this article we will discuss the basic ideas of a CSD and describe how to create and use a CSD. We will then exemplify two applications for a CSD, namely modeling co-reference in a language-specific or general fashion, and interrelating different annotations of text.

What is a CSD ?

Formal properties of a CSD

A CSD is an XML document that models a hierarchical organized set of classes given by context descriptions. In the terminology of a CSD, a context is a set of element nodes within an XML document that share some specific structural property. The hierarchy is constructed by subsetting contexts. The hierarchy of context classes requires each subclass to describe a subcontext of the superclass, i.e. the structural test performed has to be more specific. Subclasses of the same superclass are not required to form a proper decomposition, so there may be some subcontexts sharing several nodes. In our example of the `name` elements in Fig. 2, the general context description is that their ancestor is an element `s`. The first occurrence of `name` and its fourth occurrence can be described as a subclass, because they are the first child of an `s` element. The first occurrence of `name` forms another subclass, because it is at the beginning of an `s` element which is at the beginning of a paragraph `p`.

We have chosen caterpillar expressions as described by Brüggemann-Klein and Wood (2000) to formalize the structural properties which form the set of context-nodes. A caterpillar expression is a regular expression over an alphabet of symbols for moves (`left`, `right`, `up`, `firstChild`, `lastChild`), names of elements and several symbols for positional tests (`isRoot`, `isLeaf`, `isFirst`, `isLast`). Only element nodes are subject to a caterpillar expression and its evaluation. For example in Fig. 2, the fourth occurrence of `name` is the first child of `s`, so it is matched by a caterpillar expression like `isFirst`. The textual data “from where” (see Fig. 1) preceding the `name` element is not subject to the evaluation of the caterpillar expressions.

We will not give a lengthy description of the exact semantics of those expressions but will concentrate on their application to markup over textual data. The interpretation of each symbol can be grasped intuitively, when we imagine a caterpillar crawling in the element tree. The symbol `right` maps to `true` and a change of the current node of the caterpillar to the right sibling, if there exists such a right sibling. Otherwise the move evaluates to `false` and the current-node remains the same. Other moves, such as `up` or `left` are defined analogously. Element names and positional tests are Boolean predicates (e.g. `isFirst`) and check the current-node for specific properties, e.g. `isFirst` evaluates to `true`, if the current-node is the first child of its parent. A caterpillar expression evaluates to `true` with respect to some arbitrary initial node, i.e. the tested node belongs to the context, if there is a mapping of the expression to a sequence of successful moves and tests in the element tree.

Related approaches

CSD might resemble Schematron (Jelliffe, 2001), as both can be used to partially validate documents via description of permissible paths for elements, but in fact CSD differs from this approach in several aspects. First, Schematron uses XPath (Clark and DeRose, 1999) to specify the paths, which is more expressive than caterpillar expressions. Undoubtfully this eases describing contexts. However, we are not only interested in modeling contexts but also in comparing and relating context-descriptions to document grammars in order to be able to compare their strengths and weaknesses for (linguistic) modeling. Hence, less expressive languages seem to be better suited. Second, Schematron almost "only" deals with reporting fail tests, whereas CSD is especially designed for classification of nodes, i.e., to assign the set

of contexts the node belongs to. We can think of CSD as a means for weak typing as it can be found in several query languages. Certainly, one can mimic this using Schematrons named `pattern`, but at the expense of losing some level of abstraction. Nevertheless, CSD and Schematron share the capability to describe and validate documents based on an open, node-centric view instead of the top-down hierarchical approach forced by document grammars.
CSD can also be compared to the declaration of feature structures in the TEI (Sperberg-McQueen and Burnard 1994). The basic idea is the same, namely to use an additional document to specify properties of the basic data in form of constraints. Similar to Schematron, the expressive power of the TEI feature structures is much higher than that of caterpillar expressions. But, as mentioned before, for our theoretical interests in the relation between grammatical and path expressible constraints a restriction to a less expressive language seems to be worthwhile.

How to write a CSD

The structure of the CSD and the output document of processing is formulated using the DTD formalism. An instance of a CSD is fragmentary illustrated in Fig. 3:

```
<csd mode="query">
  <namespaceList>
    <namespace prefix="xxx" uri="http://www.example.com/yourNamespace" />
    ...
  </namespaceList>
  <superclass scope="element-name1 element-name2 ...">
    <class id="class-no1" sufficient="yes">
      <comment>subtype of class number 1</comment>
      <caterpillar>...</caterpillar>
    </class>
  </superclass>
  <superclass>...</superclass>
</csd>
```

Fig. 3 The general structure of a CSD given by example

A CSD is aware of namespaces, whose tuples of prefix and URI can be introduced in the `namespace` element inside the `namespaceList`. The tests for elements then may use these prefixes. As a CSD validates or queries partial document structures, we need to define only those elements that we consider relevant to the process of querying or validation (see below). This is specified by the `scope` attribute, which is attached to the `superclass` element. The value of `scope` can be a single element name or a white-space separated list of element names. A CSD that defines contexts for the element `name` (see Fig. 2) therefore contains a `superclass` element with an attribute `scope="name para ..."`.

Next we construct caterpillar expressions. A possible start node of a caterpillar is taken from the `scope` attribute. For each move or test of a caterpillar, we use the appropriate CSD element, i.e. `up`, `right`, `left`, `first`, `last`, `isRoot`, `isLeaf`, `isFirst`, `isLast`.[i] The name of an element is tested by `element name="some element name"`. For example, to test whether an element `s` is the ancestor of the `name` element, can be achieved by an expression like '`up, s`'. The CSD element `zeroOrMore` represents the Kleene-star operator, e.g. known from DTDs. The elements `right` and `name` as the content of `zeroOrMore` means "zero or more occurrences of the element `name` to the right of the current node". This caterpillar expression would match for all occurrences of the `name` element in Fig. 2.

Now we construct the hierarchy of caterpillar expressions. The classes are arranged in an inheritance structure, i.e. the tested properties of a certain class `n` are common to all subclasses nested in `n`. Figure 4 visualizes the class structure we have described so far for the `name` elements. With this CSD, we are able to classify the first occurrence of `name` as a member of the class `name-sub2`, the fourth occurrence of `name` as a member of the class

name-sub1, and the other occurrences as a member of name-general. Note that the common subsequences of the caterpillar expressions are omitted as they are implied by the class hierarchy.

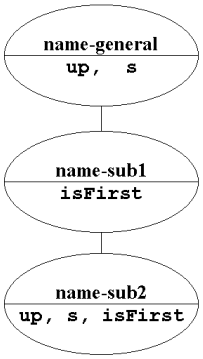

Fig. 4 The class structure of a CSD describing contextual properties of the name element in Fig. 2

How can a CSD be used?

Two constructs in the CSD in Fig. 3 have not been explained so far, i.e. the mode attribute on the csd element and the sufficient attribute on some of the class instances. These attributes are important parameters as we apply the CSD to a document instance. One can either test if a document instance is valid with respect to some context specifications, or one can query for the set of classes matched by certain element nodes. The mode attribute and its permissible values validate versus query determine the mode of processing.

Contexts (classes) can be stated to be necessary but not sufficient for validating or querying a node. The sufficient attribute is attached to a class if this class leads to a positive result in the query or validation. For example, if we are interested in querying name elements in our example document (see Fig. 1) which are at the beginning of a sentence, we would attach the sufficient attribute only to the name-sub1 class and set the mode to query. If we simply want to assure that all names are inside sentences, we would attach the sufficient attribute to the name-general class and set the mode to validate.

In some cases it might be useful to set validity constraints for certain element nodes in the document instance to ensure that a specific element matches a specific class. For this purpose we introduced the csd:caterpillar attribute defined in the namespace http://www.coli.lili.uni-bielfeld.de/projects/CSD. This attribute can be attached to elements in the document instance. The CSD-processor will generate an error for this element if it is not in accordance with the class specified in the value of csd:caterpillar. In our example, we could attach the csd:caterpillar attribute with the value name-sub2 to the first occurrence of the name element, to assure that it is always in the first sentence s.

The following list summarizes how to create and use a CSD:

- Choose one or more XML-documents to be validated / queried
- Choose an element name or a group of element names
- Write caterpillar expressions to be matched by the elements
- Construct a class hierarchy for the caterpillar expressions
- Choose classes to be sufficient for validation or query
- Write a CSD
- Optionally, attach the csd:caterpillar attribute to certain nodes in the document instance(s)
- Choose a processing-mode for the CSD, i.e. validate or query

Possible results of applying a CSD to a document instance

After processing a document instance in query mode, an output document is generated:

```
<nodelists>
  <document url="http://www.example.com/example147.xml" />
  <nodelist scope="element-name1 element-name2 ...">
    <node path="xpath-expression">
      <class name="subclass1-of-class-no1">
        <comment>This is subclass 1 of class number 1</comment>
      </class>
    </node>
  </nodelist>
</nodelists>
```

Fig. 5 The output document of a query

The output document contains a collection of `nodelist` elements, one for each superclass defined in the CSD. The `scope` attribute of each `nodelist` carries the same value as in the CSD. The location of the document instance is contained in the `url` attribute of the `document` element. Each `nodelist` consists of at least one `node`, specifying an absolute path to the respective node in the document instance. The path is expressed in XPath-Syntax, so the output document can be easily processed, e.g. with XSLT (Clark, 1999). For each sufficient class matched by the node, there is a `class` element holding the name of that class and an optional comment taken from the CSD.

The result of validating a document instance is either `true` or `false`. A document is erroneous if any node in the instance named by the `scope` attribute does not match any class regarded as `sufficient`, or if the class named by the `csd:caterpillar` attribute is not a member of the set of matching classes. That is, the corresponding caterpillar expression of the given class evaluates to `false` for that specific node. Suppose we attach the `csd:caterpillar` attribute with the value `name-sub2` to the second occurrence of `name`, then an error would occur because the expression `up s isFirst` is not true for this node.

Example applications for a CSD

Modeling of co-reference

In Sasaki et al. (2002), we present an approach towards a formal description of co-reference in different languages, using the expressive power of document grammars. There we create general and language-specific document grammars for language corpora. In this paper we will not give a detailed description of this approach, but try to exemplify how the description of element classes in contextually specified document structures might contribute to a classification of co-referential relations, complementing the approach of document grammars. Consider the example in Fig. 6, which is a slightly modified version of the example in Fig. 1:

```
<corpus>
  <p>
    <s>
      <name>Ministry of Truth</name>
      , -
      <name>Minitrue</name>
      , in
      <name>Newspeak</name>
      - was startlingly different from any other object in sight.
    </s>
    <s>
      <pron>It</pron>
      was an enormous pyramidal structure of glittering white concrete, soaring up, terrace
      after terrace, 300 metres into the air.
    </s>
    <s>
      From where
      <name>Winston</name>
      stood
      <pron>it</pron>
      was just possible to read, picked out on its white face in elegant lettering, the three
      slogans of the
      <name>Party</name>
      :
      <q>War is peace</q>
      <q>Freedom is slavery</q>
      <q>Ignorance is strength.</q>
    </s>
  </p>
</corpus>
```

Fig 6. Co-referential units

The noun phrase "Ministry of truth", tagged as `name`, co-refers with the pronoun "it", which is tagged as `pron` in the second sentence `s`. “Minitrue” also co-refers with the noun "minitrue" in the same sentence, which is tagged as `name`. In the third sentence `s`, there is another pronoun `pron` which refers to the three quotations `q`. Fig. 7 visualizes the structural properties of the three co-referential units and a corresponding CSD:

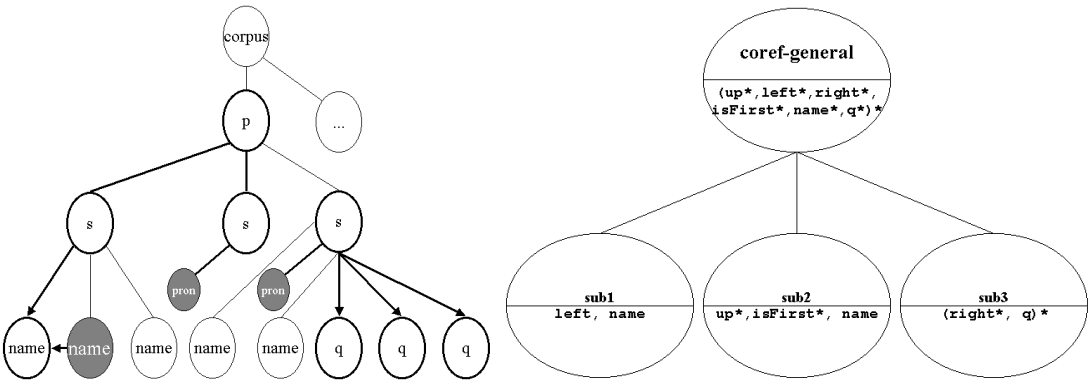

Fig. 7 Structural properties of co-referential units in Fig. 6 and a corresponding CSD

“Minitrue” can be related to “Ministry of Truth” with the caterpillar expression `left name`. The first occurrence of the pronoun `pron` can by related to “Ministry of Truth” with another caterpillar expression `up up isFirst* name`. And the second pronoun `pron` can be related to the three quotations `q` via the caterpillar expression `(right* q)*`. The visualization of the CSD shows how these structural specifications can be classified.

This example shows how one might use a CSD in the field of linguistics. It can be a starting point to describe structural properties of certain co-referential phenomena and to

allow to test them with annotated textual data beyond the practical limitations of document grammars in a general and more or less (language) specific fashion.

Interrelating different annotations of text

There has been a long ongoing discussion on how to represent concurrent hierarchies in document structures. One source of this discussion is the OHCO hypothesis that text is a ordered hierarchy of content objects. There are many weak and strong versions of this hypothesis. Some authors (Caton, 2002) even claim that the idea of text as a hierarchical structure is just one plausible view among others.

We do not claim to be able to contribute new ideas for this discussion or a solution for the problem. What we want to try is to interrelate different annotations with a CSD. We represent one primary annotation in the ordinary XML document structure and another annotation, in the same document, with anchor elements. A type can be assigned to these anchors via the `csd:caterpillar` attribute, and the respective classes in the CSD can specify the structural constraints for the anchors. Fig. 8 shows an example of a primary annotation from a linguistic perspective which marks sentences with tag `s` and a secondary annotation which marks lines with `line-begin` and `line-end`:

```
- <corpus xmlns:csd="www.text-technology.de/projects/csd">
    <line-begin csd:caterpillar="firstLine" />
  - <s>
      The Ministry of Truth - Minitrue, in Newspeak* - was
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      startlingly different from any other object in sight.
    </s>
  - <s>
      It was
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      an enormous pyramidal structure of glittering white
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      concrete, soaring up, terrace after terrace, 300 metres into
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      the air.
    </s>
  - <s>
      From where Winston stood it was just possible to
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      read, picked out on its white face in elegant lettering, the
      <line-end />
      <line-begin csd:caterpillar="normalLine" />
      three slogans of the Party:
    </s>
    <line-end />
    <line-begin csd:caterpillar="identicalToSentence" />
    <s>WAR IS PEACE</s>
    <line-end />
    <line-begin csd:caterpillar="identicalToSentence" />
    <s>FREEDOM IS SLAVERY</s>
    <line-end />
    <line-begin csd:caterpillar="identicalToSentence" />
    <s>IGNORANCE IS STRENGTH</s>
    <line-end />
  </corpus>
```

Fig. 8 An annotation of lines and sentences

With the CSD, it is possible to specify different types of relations between the annotation of sentences `s` and lines. We can define a class `normalLine` for general lines,

which follow immediately a `line-end` element. The respective caterpillar expression for this class is `left line-end`. The subclass `last-line-begin` has the caterpillar expression `right* line-end isLast up corpus`. There is also a class which cannot be subsumed under the `normalLine class`. This class is called `identicalToSentence` and has the caterpillar expression `isLast up s`.

This methodology is closely related to solutions for the problem of overlapping hierarchies, as proposed by the TEI. One of the TEI solutions is the construction of virtual joints for fragmentary elements. A CSD can be used for this purpose as well, but also – as in our example – to describe various classes for the instances of the 'secondary' annotation. These then can be used to test hypothesis about the relations between two different annotations of text.

Still our methodology has some drawbacks, especially the fact that so far it is not yet possible to generate the caterpillar expressions automatically. Nevertheless, it can be used to validate a hypothesis about the relations between different annotations of the same textual data.

Summary and future work

In this article, we have described the motivation for the contextual specification of elements and their representation in a class structure. We presented the main aspects of CSD as a framework and some examples. Two applications in the domain of co-reference and the modeling of different annotations of text showed the potential of CSD.

A prototype of a CSD processor has been implemented in the Python programming language. In the future we will continue research on several subjects. As described, currently we follow Brüggemann-Klein and Wood in restricting the operators to sequence, brackets and Kleene-star. However, we expect optionality '?' to be highly valuable to impose locality constraints (e.g. at most three nodes to the `left? left? left?`) and therefore consider to extend our notion of caterpillar expressions in that sense. Since CSD uses but does not depend on a specific path language, it is easy to integrate for example XPath or other (path) languages with minor modifications to the CSD-DTD, as path expressions may be given as attribute values as well. Furthermore, we want to explore in more detail the relation between a CSD based on caterpillar expression and document grammars. And last but not least - we want to use the CSD to model linguistic phenomena on large corpora as another approach to secondary information structuring.

References

Bray, T., J. Paoli, C. M. Sperberg-McQueen and Eve Maler (2000). Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation 6 October 2000. See http://www.w3.org/TR/REC-xml

Brüggemann-Klein, A and D. Wood (2000). Caterpillars: A Context Specification Technique, Markup Languages: Theory & Practice, 2(1):81-106.

Caton, P. (2002). Markup's Current Imbalance Markup Languages: Theory & Practice, 3(1):1-13.

Clark, J. (1999). XSL Transformations (XSLT). W3C Recommendations 16 November 1999. See http://www.w3.org/TR/xslt

Clark, J. and S. DeRose (1999). XML Path Language (XPath). W3C Recommendation 16 November 1999. See http://www.w3.org/TR/xpath

Clark, J. and M. Murata (2001). Relax NG Specification. OASIS Committee Specification 3 December 2001. See http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

Ide, N. and J. Véronis (1994). Multext (multilingual tools and corpora), Proceedings of the 15th CoLing, Kyoto, 90-96.

Jelliffe, R. (2001). Schematron – an XML Structure Validation Language using Patterns in Trees. See http://www.ascc.net/xml/schematron/
Sasaki, F., C. Wegener, A. Witt, D. Metzing and J. Pönninghaus (2002). Co-reference annotation and resources: a multilingual corpus of typologically diverse languages, Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC-2002), Las Palmas, 1225-1231.
Sperberg-McQueen, M. and L. Burnard (eds). Guidelines for Electronic Text Encoding and Interchange (TEI P3), ACH / ACL / ALLC, Chicago / Oxford, 1994.
Thompson, H., D. Beech, M. Maloney and N. Mendelsohn (2001). XML Schema Part 1: Structures. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/xmlschema-1

[i] Experienced users may give path expressions as attribute values using concise notation.