Testing Structural Properties in Textual Data: Beyond Document Grammars

Felix Sasaki and Jens Pönninghaus
University of Bielefeld, Germany

Abstract

Schema languages concentrate on grammatical constraints on document structures, i.e. hierarchical relations between elements in a tree-like structure. In this paper, we complement this concept with a methodology for defining and applying structural constraints from the perspective of a single element. These constraints can be used in addition to the existing constraints of a document grammar. There is no need to change the document grammar. Using a hierarchy of descriptions of such constraints allows for a classification of elements. These are important features for tasks such as visualizing, modelling, querying, and checking consistency in textual data. A document containing descriptions of such constraints we call a 'context specification document' (CSD). We describe the basic ideas of a CSD, its formal properties, the path language we are currently using, and related approaches. Then we show how to create and use a CSD. We give two example applications for a CSD. Modelling co-referential relations between textual units with a CSD can help to maintain consistency in textual data and to explore the linguistic properties of co-reference. In the area of textual, non-hierarchical annotation, several annotations can be held in one document and interrelated by the CSD. In the future we want to explore the relation and interaction between the underlying path language of the CSD and document grammars.

1 Introduction

This paper describes research carried out in the project 'Secondary information structuring and comparative discourse analysis' (SEKIMO), which is part of the research group 'Text-technological modelling 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, i.e. morpho-syntactic structures, semantic relations, and discourse functions, and in the granularity of the description, i.e. there are language or dialogue

Correspondence:
Felix Sasaki, University of Bielefeld, Faculty of Linguistics and Literature, Computational Linguistics and Text-Technology, 33501 Bielefeld, Germany.
E-mail:
felix.sasaki@uni-bielefeld.de

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 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 that 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 fulfilment of constraints.

We will exemplify this in reference to the document shown in Fig. 1, which is based on the English part of the MULTEXT-EAST corpus (Ide and Véronis, 1994).

```
<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 shown in Fig. 2.

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`, shown by arrows pointing from `name` elements to `s` elements. This feature is shared by all `name` elements. Other features are shared by only 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`, shown by the dark 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 that define more specific properties.

For tasks such as visualizing, modelling, querying, and checking consistency of text, it might be very useful to describe the contextual features of elements and arrange them in a class structure. A document containing such descriptions we call a 'context specification document' (CSD). In this paper we will discuss the basic ideas of a CSD and describe how to create and use a CSD. We will then give two example applications for a CSD, namely modelling co-reference in a language-specific or general fashion, and interrelating different annotations of text.

2 What is a CSD?

2.1 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

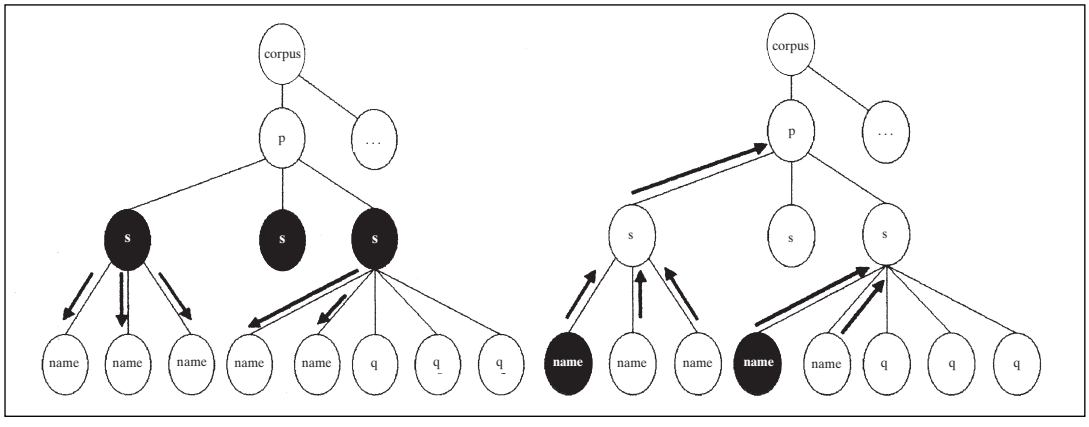

Fig. 2 Hierarchical and contextual constraints on elements.

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 that 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 that 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 such as `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 these 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.

2.2 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. Undoubtedly this eases describing contexts. However, we are interested not only in modelling contexts but also in comparing and relating context-descriptions to document grammars so as to be able to compare their

strengths and weaknesses for (linguistic) modelling. 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 with the declaration of feature structures in the Text Encoding Initiative (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. However, as mentioned above, for our theoretical interests in the relation between grammatical and path expressible constraints a restriction to a less expressive language seems to be worth while.

2.3 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 illustrated in Fig. 3.

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

```
<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.

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`.[1] The name of an element is tested by `element name="some element name"`. For example, we can test whether an element `s` is the ancestor of the `name` element by using an expression such as `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` mean '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 shows 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`. It should be noted that the common subsequences of the caterpillar expressions are omitted as they are implied by the class hierarchy.

2.4 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. We can either test if a document instance is valid with respect to some context specifications or 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

1 Experienced users may give path expressions as attribute values using concise notation.

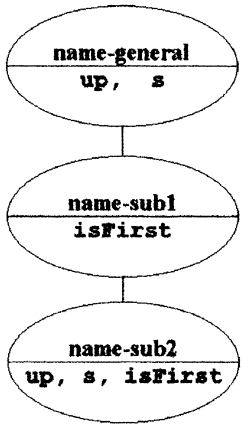

Fig. 4 The class structure of a CSD describing contextual properties of the `name` element in Fig. 2.

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) that 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 ensure 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.text-technology.de/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 ensure 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 or 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`.

2.5 Possible results of applying a CSD to a document instance

After processing a document instance in `query` mode, an output document is generated (Fig. 5).

The output document contains a collection of `nodelist` elements, one for each superclass defined in the CSD. The `scope` attribute of each

```
<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.

`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.

3 Example Applications for a CSD

3.1 Modelling 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. Let us consider the example in Fig. 6, which is a slightly modified version of the example in Fig. 1.

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`. Figure 7 shows the structural properties of the three co-referential units 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

```
<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.

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

3.2 Interrelating different annotations of text

There has been a long continuing 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 workers (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. Figure 8 shows an example of a primary annotation from a linguistic

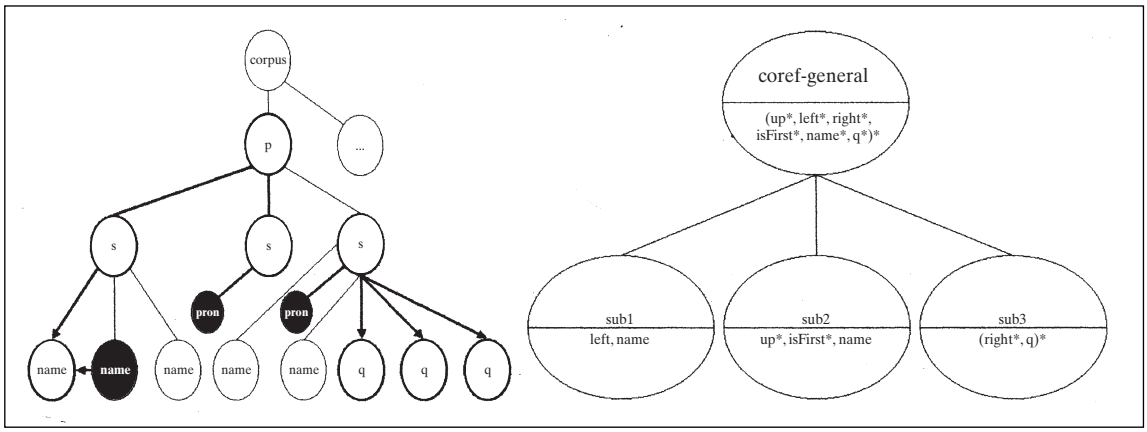

Fig. 7 Structural properties of co-referential units in Fig. 6 and a corresponding CSD.

perspective, which marks sentences with tag `s`, and a secondary annotation, which marks lines with `line-begin` and `line-end`.

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 that 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 hypotheses about the relations between two different annotations of text.

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.

4 Summary and Future Work

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

```
<corpus xmlns:csd="www.text-technology.de/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.

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 extending our notion of caterpillar expressions in that sense. As 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., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. (2000). Extensible Markup Language (XML) 1.0, 2nd edn. W3C Recommendation, 6 October 2000. http://www.w3.org/TR/REC-xml (accessed 6 April 2003).

Brüggemann-Klein, A. and Wood, D. (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. http://www.w3.org/TR/xslt (accessed 6 April 2003).

Clark, J. and DeRose, S. (1999). XML Path Language (XPath). W3C Recommendation, 16 November 1999. http://www.w3.org/TR/xpath (accessed 6 April 2003).

Clark, J. and Murata, M. (2001). Relax NG Specification. OASIS Committee Specification, 3 December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html (accessed 6 April 2003).

Ide, N. and Véronis, J. (1994). Multext (multilingual tools and corpora). *Proceedings of the 15th CoLing,* Kyoto. Sheffield: ICCL (International Committee on Computational Linguistics), pp. 90–6.

Jelliffe, R. (2001). Schematron—an XML structure validation language using patterns in trees. http://www.ascc.net/xml/schematron/ (accessed 6 April 2003).

Sasaki, F., Wegener, C., Witt, A., Metzing, D., and Pönninghaus, J. (2002). Coreference 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. Paris: ELRA (European Language Resources Association), pp. 1225–31.

Sperberg-McQueen, M. and Burnard, L. (eds) (1994). Guidelines for Electronic Text Encoding and Interchange (TEI P3). Chicago/Oxford: ACH/ACL/ALLC.

Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. (2001). XML Schema Part 1: Structures. W3C Recommendation, 2 May 2001. http://www.w3.org/TR/xmlschema-1 (accessed 6 April 2003).