Towards A Modular Data Model For Multi-Layer Annotated Corpora

Richard Eckart
Department of English Linguistics
Darmstadt University of Technology
64289 Darmstadt, Germany
eckart@linglit.tu-darmstadt.de

Abstract

In this paper we discuss the current methods in the representation of corpora annotated at multiple levels of linguistic organization (so-called *multi-level* or *multi-layer* corpora). Taking five approaches which are representative of the current practice in this area, we discuss the commonalities and differences between them focusing on the underlying data models. The goal of the paper is to identify the common concerns in multi-layer corpus representation and processing so as to lay a foundation for a unifying, modular data model.

1 Introduction

Five approaches to representing multi-layer annotated corpora are reviewed in this paper. These reflect the current practice in the field and show the requirements typically posed on multi-layer corpus applications. Multi-layer annotated corpora keep annotations at different levels of linguistic organization separate from each other. Figure 1 illustrates two annotation layers on a transcription of an audio/video signal. One layer contains a functional annotation of a sentence in the transcription. The other contains a phrase structure annotation and Part-of-Speech tags for each word. Layers and signals are coordinated by a common timeline.

The motivation for this research is rooted in finding a proper data model for PACE-Ling (Sec. 2.2). The ultimate goal of our research is to create a modular extensible data model for multi-layer annotated corpora. To achieve this, we aim to create a data model based on the current state-of-the-art that covers all current requirements and

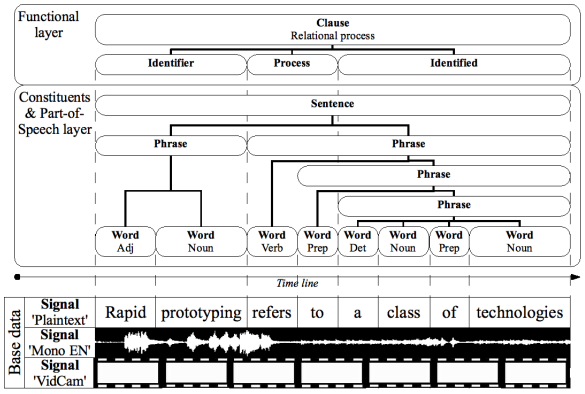

Figure 1: Multi-layer annotation on multi-modal base data

then decompose it into exchangeable components. We identify and discuss objects contained in four tiers commonly playing an important role in multi-layer corpus scenarios (see Fig. 2): *medial*, *locational*, *structural* and *featural* tiers. These are generalized categories that are in principle present in any multi-layer context, but come in different incarnations. Since query language and data model are closely related, common query requirements are also surveyed and examined for modular decomposition. While parts of the suggested data model and query operators are implemented by the projects discussed here, so far no comprehensive implementation exists.

2 Data models

There are three purposes data models can serve. The first purpose is *context suitability*. A data model used for this purpose must reflect as well as possible the data the user wants to query. The second purpose is *storage*. The data model used in the database backend can be very different from

Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions, pages 183–190,
Sydney, July 2006.

the one exposed to the user, e.g. hierarchical structures may be stored in tables, indices might be kept to speed up queries, etc. The third purpose is *exchange* and *archival*. Here the data model, or rather the serialization of the data model, has to be easily parsable and follow a widely used standard.

Our review focuses on the suitability of data models for the first purpose. As extensions of the XML data model are used in most of the approaches reviewed here, a short introduction to this data model will be given first.

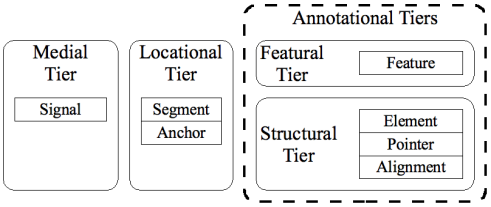

Figure 2: Tiers and objects

2.1 XML

Today XML has become the de-facto standard representation format for annotated text corpora. While the XML standard specifies a data model and serialization format for XML, a semantics is largely left to be defined for a particular application. Many data models can be mapped to the XML data model and serialized to XML (cf. Sec. 2.5).

The XML data model describes an ordered tree and defines several types of nodes. We examine a simplification of this data model here, limited to *elements*, *attributes* and *text nodes*. An element (*parent*) can contain *children*: elements and text nodes. Elements are named and can carry attributes, which are identified by a name and bear a value.

This data model is immediately suitable for simple text annotations. For example in a positional annotation, name-value pairs (*features*) can be assigned to tokens, which are obtained via tokenization of a text. These features and tokens can be represented by attributes and text nodes. The XML data model requires that both share a parent element which binds them together. Because the XML data model defines a tree, an additional root element is required to govern all positional annotation elements.

If the tree is constructed in such a way that one particular traversal strategy yields all tokens in their original order, then the data model is capable of covering all tiers: medial tier (textual base data), locational tier (sequential token order), structural tier (tokens) and featural tier (linguistic feature annotations). The structural tier can be expanded by adding additional elements en-route from the root element to the text nodes (*leaves*). In this way hierarchical structures can be modeled, for instance constituency structures. However, the XML data model covers these tiers only in a limited way. For example, tokens can not overlap each other without destroying the linear token order and thus sacrificing the temporal tier, a problem commonly known as *overlapping hierarchies*.

2.2 PACE-Ling

PACE-Ling (Bartsch et al., 05) aims at developing register profiles of texts from mechanical engineering (domain: data processing in construction) based on the multi-dimensional model of Systemic Functional Linguistics (SFL) (Halliday, 04).

The XML data model is a good foundation for this project as only written texts are analyzed, but SFL annotation requires multiple annotation layers with overlapping hierarchies. To solve this problem, the project applies a strategy known as *stand-off annotation*, first discussed in the context of SFL in (Teich et al., 05) and based on previous work by (Teich et al., 01). This strategy separates the annotation data from the base data and introduces references from the annotations to the base data, thus allowing to keep multiple layers of annotations on the same base data separate.

The tools developed in the project treat annotation data in XML from any source as separate annotation layers, provided the text nodes in each layer contain the same base data. The base data is extracted and kept in a text file and the annotation layers each in an XML file. The PACE-Ling data model substitutes text nodes from the XML data model by *segments*. Segments carry *start* and *end* attributes which specify the location of the text in the text file.

An important aspect of the PACE-Ling approach is minimal invasiveness. The minimally invasive change of only substituting text nodes by segments and leaving the rest of the original annotation file as it is, makes conversion between the original format and the format needed by the PACE-Ling tools very easy.

2.3 NITE XML Toolkit

The NITE XML toolkit (NXT) (Carletta et al., 04) was created with the intention to provide a framework for building applications working with annotated multi-modal data. NXT is based on the NITE Object Model (NOM) which is an extension of the XML data model. NOM features a similar separation of tiers as the PACE-Ling data model, but is more general.

NOM uses a continuous *timeline* to coordinate annotations. Instead of having dedicated segment elements, any annotation element can have special *start* and *end* attributes that anchor it to the timeline. This makes the data model less modular, because support for handling other locational strategies than a timeline can not be added by changing the semantics of segments (cf. Sec. 3.2).

NXT can deal with audio, video and textual base data, but due to being limited to the concept of a single common timeline, it is not possible to annotate a specific region in one video frame.

NOM introduces a new structural relation between annotation elements. Arbitrary links can be created by adding a *pointer* to an annotation element bearing a reference to another annotation element which designates the first annotation element to be a parent of the latter. Each pointer carries a role attribute describing its use.

Using pointers, arbitrary directed graphs can be overlaid on annotation layers and annotation elements can have multiple parents, one from the layer structure and any number of parents indicated by pointer references. This facilitates the reuse of annotations, e.g. when a number of annotations are kept that apply to words, the boundaries of words can be defined in one annotation layer and the other annotations can refer to that via pointers instead of defining the word boundaries explicitly in each layer. Using these pointers in queries is cumbersome, because they have to be processed one at a time (Evert et al., 03).

2.4 Deutsch Diachron Digital

The goal of *Deutsch Diachron Digital* (DDD) (Faulstich et al., 05) is the creation of a diachronic corpus, ranging from the earliest Old High German or Old Saxon texts from the 9th century up to Modern German at the end of the 19th century.

DDD requires each text to be available in several versions, ranging from the original facsimile over several transcription versions to translations into a modern language stage. This calls for a high degree of alignment between those versions as well as the annotations on those texts. Due to the vast amount of data involved in the project, the data model is not mapped to XML files, but to a SQL database for a better query performance.

The DDD data model can be seen as an extension of NOM. Because the corpus contains multiple versions of documents, coordination of annotations and base data along a single timeline is not sufficient. Therefore DDD segments refer to a specific version of a document.

DDD defines how *alignments* are modeled, thus elevating them from the level of structural annotation to an independent object in the structural tier: an alignment as a set of elements or segments, each of which is associated with a role.

Treating alignments as an independent object is reasonable because they are conceptually different from pointers and it facilitates providing an efficient storage for alignments.

2.5 ATLAS

The ATLAS project (Laprun et al., 02) implements a three tier data model model, resembling the separation of medial, locational and annotation tiers. This approach features two characteristic traits setting it apart from the others. First the data model is not inspired by XML, but by Annotation Graphs (AGs) (Bird & Liberman, 01). Second, it does not put any restriction on the kind of base data by leaving the semantics of segments and anchors undefined.

The ATLAS data model defines *signals*, *elements*, *attributes*, *pointers*, *segments* and *anchors*. Signals are base data objects (text, audio, etc.). Elements are related to each other only using pointers. While elements and pointers can be used to form trees, the ATLAS data model does not enforce this. As a result, the problem of overlapping hierarchies does not apply to the model. Elements are not contained within layers, instead they carry a type. However all elements of the same type can be interpreted as belonging to one layer. Segments do not carry start and end attributes, they carry a number of anchors. How exactly anchors are realized depends on the signals and is not specified in the data model.

The serialization format of ATLAS (AIF) is an XML dialect, but does not use the provisions for modeling trees present in the XML data model to

represent structural annotations as e.g. NXT does. The annotation data is stored as a flat set of elements, pointers, etc., which precludes the efficient use of existing tools like XPath to do structural queries. This is especially inconvenient as the ATLAS project does not provide a query language and query engine yet.

2.6 ISO 24610-1 - Feature Structures

The philosophy behind (ISO-24610-1, 06) is different from that of the four previous approaches. Here the base data is an XML document conforming to the TEI standard (Sperberg-McQueen & Burnard, 02). XML elements in the TEI base data can reference *feature stuctures*. A feature structure is a single-rooted graph, not necessarily a tree. The inner nodes of the graph are typed *elements*, the leaves are *values*, which can be shared amongst elements using *pointers* or can be obtained functionally from other values.

While in the four previously discussed approaches the annotations contain references to the base data in the leaves of the annotation structure, here the base data contains references to the root of the annotation structures. This is a powerful approach to identifying features of base data segments, but it is not very well suited for representing constituent hierarchies.

Feature structures put a layer of abstraction on top of the facilities provided by XML. XML validation schemes are used only to check the well-formedness of the serialization but not to validate the features structures. For this purpose *feature structure declarations* (FSD) have been defined.

3 A comprehensive data model

This section suggests a data model covering the objects that have been discussed in the context of the approaches presented in Sections 2.1-2.6. See Figure 3 for an overview.

3.1 Objects of the medial tier

We use the term *base data* for any data we want to annotate. A single instance of base data is called *signal*. Signals can be of many different kinds such as images (e.g. scans of facsimiles) or streams of text, audio or video data.

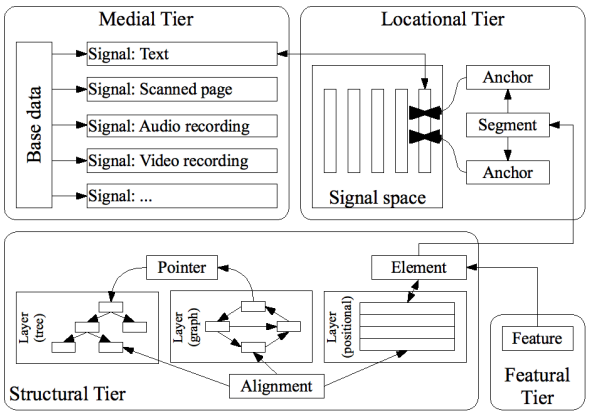

Figure 3: Comprehensive data model

3.2 Objects of the locational tier

Signals live in a virtual multi-dimensional *signal space*[1]. Each point of a signal is mapped to a unique point in signal space and vice versa. A *segment* identifies an area of signal space using a number of *anchors*, which uniquely identify points in signal space.

Depending on the kind of signal the dimensions of signal space have to be interpreted differently. For instance streams have a single dimension: time. At each point along the time axis, we may find a character or sound sample. Other kinds of signals can however have more dimensions: height, width, depth, etc. which can be continuous or discrete, bounded or open. For instance, a sheet of paper has two bounded and continuous dimensions: height and width. Thus a segment to capture a paragraph may have to describe a polygon. A single sheet of paper does not have a time dimension, however when multiple sheets are observed, these can be interpreted as a third dimension of discrete time.

3.3 Objects of the annotational tiers

An annotation *element* has a name and can have *features*, *pointers* and *segments*. A pointer is a typed directed reference to one or more elements. Elements relate to each other in different ways: directly by structural relations of the layer, pointers and alignments and indirectly by locational and medial relations (cf. Fig. 4).

An annotation *layer* contains elements and defines structural relations between them, e.g. *dominance* or *neighborhood* relations.

[1] (Laprun et al., 02) calls this *feature space*. This label is not used here to avoid suggesting a connection to the featural tier.

An *alignment* defines an equivalence class of elements, to each of which a *role* can be assigned.

Pointers can be used for structural relations that cross-cut the structural model of a layer or to create a relation across layer boundaries. Each pointer carries a role that specifies the kind of relation it models. Pointers allow an element to have multiple parents and to refer to other elements across annotation layers.

Features have a name and a value. They are always bound to an annotation element and cannot exist on their own. For the time being we use this simple definition of a feature, as it mirrors the concept of XML attributes. However, future work has to analyze if the ISO 24610 feature structures can and should be modelled as a part of the structural tier or if the featural tier should be extended.

4 Query

To make use of annotated corpora, query methods need to be defined. Depending on the data storage model that is used, different query languages are possible, e.g. XQuery for XML or SQL for relational databases. But these complicate query formulating because they are tailored to query a low level data storage model rather than a high level annotation data model.

A high level query language is necessary to get a good user acceptance and to achieve independence from lower level data models used to represent annotation data in an efficient way. NXT comes with NQL (Evert et al., 03), a sophisticated declarative high level query language. NQL is implemented in a completely new query engine instead of using XPath, XQuery or SQL. LPath, another recent development (Bird et al., 06), is a path-like query language. It is a linguistically motivated extension of XPath with additional axes and operators that allow additional queries and simplify others.

In some cases XML or SQL databases are simply not suited for a specific query. While we might be able to do regular expression matches on textual base data in a SQL or XML environment, doing a similar operation on video base data is beyond their scope.

The NXT project plans a translation of NQL to XQuery in order to use existing XQuery engines. LPath and DDD map high level query languages to SQL. (Grust et al., 04) are working on translating XQuery to SQL. The possibility of translating high level query languages into lower level query languages seems a good point for modularization.

4.1 Structural queries

Structural query operators are strongly tied to the structure of annotation layers, because they reflect the structural relations inside a layer. However, we also define structural relations such as alignments and pointers that exist independently of layers (cf. Sec. 3.3). The separation between pointers, alignments and different kinds of layers offers potential for modularization

Layers allowing only for positional annotations know only one structural relation: the neighborhood relation between two adjacent positions. Layers following the XML data model know parent-child relations and neighborhood relations. Layers with different internal structures may offer other relations. A number of possible relations is shown in Figure 4.

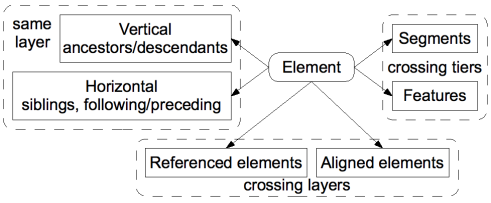

Figure 4: Structural relations and crossing to other tiers

While the implementation of query operators depends on the internal layer structure, the syntax does not necessarily have to be different. For instance a $following(a)$ operator of a positional layer will yield all elements following element a. A hierarchical layer can have two kinds of $following$ operators, one that only yields siblings following a and one yielding all elements following a. Here a choice has to be made if one of these operators is similar enough to the $following(a)$ to share that name without confusing the user.

Operators to follow pointers or alignments can be implemented independently of the layer structure.

XPath or LPath (Bird et al., 06) are path-like query languages specifically suited to access hierarchically structured data, but neither directly supports alignments, pointers or the locational tier. In the context of XQuery, XPath can be extended with user-defined functions that could be used to provide this access, but using such functions in path statements can become awkward. It may be a better idea to extend the path language instead.

Structural queries could look like this:

- Which noun phrases are inside verb phrases?
  `//VP//NP`
  Result: a set of annotation elements.

- Anaphora are annotated using a pointer with the role "anaphor". What do determiners in the corpus refer to?
  `//DET/=>anaphor`
  Result: a set of annotation elements.

- Translated elements are aligned in an alignment called "translation". What are the translations of the current element?
  `self/#translation`
  Result: a set of annotation elements.

4.2 Featural queries

If we use the simple definition of features from Section 3.3, there is only one operator native to the featural tier that can be used to access the annotation element associated with a feature. If we use the complex definition from ISO 24610, the operators of the featural tier are largely the same as in hierarchically structured annotation layers.

Operators to test the value of a feature can not strictly be assigned to the featural tier. Using the simple definition, the value of a feature is some typed atomic value. The query language has to provide generic operators to compare atomic values like strings or numbers with each other. E.g. XPath provides a weakly typed system that provides such operators.

Queries involving features could look like this:

- What is the value of the "PoS" feature of the current annotation element?
  `self/@PoS`
  Result: a string value.

- What elements have a feature called "PoS" with the value "N"?
  `//*[@PoS='N']`
  Result: a set of annotation elements.

4.3 Locational queries

Locational queries operate on segment data. The inner structure of segments reflects the structure of signal space and different kinds of signals require different operators. Most of the time operators working on single continuous dimensions, e.g. a timeline, will be used. An operator working on higher dimensions could be an intersection operator of two dimensional signal space areas (scan of a newspaper page, video frames, etc.).

Queries involving locations could look like this:

- What parts of segments a and b overlap?
  `overlap($a,$b)`
  Result: the empty set or a segment defining the overlapping part.

- Merge segments a and b.
  `merge($a, $b)`
  Result: if a and b overlap, the result is a new segment that covers both, otherwise the results is a set consisting of a and b.

- Is segment a following segment b?
  `is-following($a, $b)`
  Result: true or false.

Locational operators are probably best bundled into modules by the kind of locational structure they support: a module for sequential data such as text or audio, one for two-dimensional data such as pictures, and so on.

4.4 Medial queries

Medial query operators access base data, but often they take locational arguments or return locational information. When a medial operator is used to access textual base data, the result is a string. As with feature values, such a string could be evaluated by a query language that supports some primitive data types.

Assume there is a textual signal named 'plaintext'. Queries on base data could look like this:

- Where does the string "rapid" occur?
  `signal('plaintext')/'rapid'`
  Result: a set of segments.

- Where does the string "prototyping" occur to the right of the location of "rapid"?
  ```
  signal('plaintext')/
      'rapid'>>'prototyping'
  ```
  Result: a set of segments.

- What is the base data between offset 5 and 9 of the signal "plaintext"?
  `signal('plaintext')/<{5,9}>`
  Result: a portion of base data (e.g. a string).

If the base data is an audio or video stream, the type system of most query languages is likely to

be insufficient. In such a case a module providing support for audio or video storage should also provide necessary query operators and data type extensions to the query engine.

4.5 Projection between annotational and medial tiers

So far we have considered crossing the borders between the structural and featural tiers and between the locational and medial tiers. Now we examine the border between the locational and structural tier. An operator can be used to collect all locational data associated with an annotation element and its children:

```
seg(//S/VP/)
```

The result would be a set of potentially overlapping segments. Depending on the query, it will be necessary to merge overlapping segments to get a list of non-overlappping segments. Assume we have a recorded interview annotated for speakers and at some point speaker A and B speak at the same time. We want to listen to all parts of the interview in which speakers A or B speak. If we query without merging overlapping segments, we will hear the part in which both speak at the same time twice.

Similar decisions have to be made when projecting up from a segment into the structural layer. Figure 5 shows a hierarchical annotation structure. Only the elements $W1$, $W2$ and $W3$ bear segments that anchor them to the base data at the points A-D.

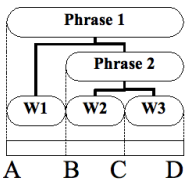

Figure 5: Example structure

When projecting up from the segment $\{B, D\}$ there are a number of potentially desirable results. Some are given here:

1. no result: because there is no annotation element that is anchored to $\{B, D\}$.

2. $W2$ and $W3$: because both are anchored to an area inside $\{B, D\}$.

3. *Phrase 2*, $W2$ and $W3$: because applying the `seg` operator to either element yields segments inside $\{B, D\}$.

4. *Phrase 2* only: because applying the `seg` operator to this element yields an area that covers exactly $\{B, D\}$.

5. *Phrase 1*, *Phrase 2*: because applying the `seg` operator to either element yields segments containing $\{B, D\}$.

The query language has to provide operators that enable the user to choose the desired result. Queries that yield the desired results could look like in Figure 6. Here the *same-extent* operator takes two sets of segments and returns those segments that are present in both lists and have the same start and end positions. The *anchored* operator takes an annotation element and returns *true* if the element is anchored. The *contains* operator takes two sets of segments a and b and returns all segments from set b that are contained in an area covered by any segment in set a. The *grow* operator takes a set of segments and returns a segment, which starts at the smallest offset and ends at the largest offset present in any segment of the input list. In the tests an empty set is interpreted as *false* and a non-empty set as *true*.

1. `//*[same-extent(seg(.), <{B,D}>)]`

2. `//*[anchored(.) and contains(<{B,D}>, seg(.))]`

3. `//*[contains(<{B,D}>, seg(.))]`

4. `//*[same-extent(grow(seg(.)), <{B,D}>)]`

5. `//*[contains(seg(.)), <{B,D}>]`

Figure 6: Projection examples

5 Conclusion

Corpus-based research projects often choose to implement custom tools and encoding formats. Small projects do not want to lose valuable time learning complex frameworks and adapting them to their needs. They often employ a custom XML format to be able to use existing XML processing tools like XQuery or XSLT processors.

ATLAS or NXT are very powerful, yet they suffer from lack of accessibility to programmers who have to adapt them to project-specific needs. Most specialized annotation editors do not build upon these frameworks and neither offer conversion tools between their data formats.

Projects such as DDD do not make use of the frameworks, because they are not easily extensible, e.g. with a SQL backend instead of an XML storage. Instead, again a high level query language is developed and a completely new framework is created which works with a SQL backend.

In the previous sections, objects from selected approaches with different foci in their work with annotated corpora have been collected and forged into a comprehensive data model. The potential for modularization of corpus annotation frameworks has been shown with respect to data models and query languages. As a next step, an existing framework should be taken and refactored into an extensible modular architecture. From a practical point of view reusing existing technology as much as possible is a desirable goal. This means reusing existing facilities provided for XML data, such as XPath, XQuery and XSchema and where necessary trying to extend them, instead of creating a new data model from scratch. For the annotational tiers, as LPath has shown, a good starting point to do so is to extend existing languages like XPath. Locational and medial operators seem to be best implemented as XQuery functions. The possibility to map between SQL and XML provides access to additional efficient resources for storing and querying annotation data. Support for various kinds of base data or locational information can be encapsulated in modules. Which modules exactly should be created and what they should cover in detail has to be further examined.

Acknowledgements

Many thanks go to Elke Teich and Peter Fankhauser for their support. Part of this research was financially supported by *Hessischer Innovationsfonds* and PACE (Partners for the Advancement of Collaborative Engineering Education).

References

S. Bartsch, R. Eckart, M. Holtz & E. Teich 2005. Corpus-based register profiling of texts form mechanical engineering In *Proceedings of Corpus Linguistics*, Birmingham, UK, July 2005.

S. Bird & M. Liberman 2001. A Formal Framework for Linguistic Annotation In *Speech Communication 33(1,2), pp 23-60*

S. Bird, Y. Chen, S. B. Davidson, H. Lee and Y. Zheng. 2006. Designing and Evaluating an XPath Dialect for Linguistic Queries. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA

J. Carletta, D. McKelvie, A. Isard, A. Mengel, M. Klein & M.B. Møller 2004 A generic approach to software support for linguistic annotation using XML In *G. Sampson and D. McCarthy (eds.), Corpus Linguistics: Readings in a Widening Discipline*. London and NY: Continuum International.

S. Evert, J. Carletta, T. J. O'Donnell, J. Kilgour, A. Vögele & H. Voormann 2003. *The NITE Object Model* v2.1 http://www.ltg.ed.ac.uk/NITE/documents/NiteObjectModel.v2.1.pdf

L. C. Faulstich, U. Leser & A. Lüdeling 2005. Storing and querying historical texts in a relational database In *Informatik-Bericht 176*, Institut für Informatik, Humboldt-Universität zu Berlin, 2005.

T. Grust and S. Sakr and J. Teubner 2002. XQuery on SQL Hosts In *Proceedings of the 30th Int'l Conference on Very Large Data Bases (VLDB)* Toronto, Canada, Aug. 2004.

M.A.K. Halliday. 2004. *Introduction to Functional Grammar*. Arnold, London. Revised by CMIM Matthiessen

C. Laprun, J.G. Fiscus, J. Garofolo, S. Pajot 2002. A practical introduction to ATLAS In *Proceedings LREC 2002* Las Palmas http://www.nist.gov/speech/atlas/download/lrec2002-atlas.pdf

M. Laurent Romary (chair) and TC 37/SC 4/WG 2 2006. Language resource management - Feature structures - Part 1: Feature structure representation. In *ISO 24610-1*.

C. M. Sperberg-McQueen & L. Burnard, (eds.) 2002. *TEI P4: Guidelines for Electronic Text Encoding and Interchange*. Text Encoding Initiative Consortium. XML Version: Oxford, Providence, Charlottesville, Bergen

E. Teich, P. Fankhauser, R. Eckart, S. Bartsch, M. Holtz. 2005. Representing SFL-annotated corpora. In *Proceedings of the First Computational Systemic Functional Grammar Workshop (CSFG)*, Sydney, Australia.

E. Teich, S. Hansen, and P. Fankhauser. 2001. Representing and querying multi-layer corpora. In *Proceedings of the IRCS Workshop on Linguistic Databases*, pages 228-237, University of Pennsylvania, Philadelphia, 11-13 December.