# A Generic XML-Based Format for Structured Linguistic Annotation and Its Application to the Prague Dependency Treebank 2.0

**Petr Pajas**
**Jan Štěpánek**

**Technical Report Edition, TR-2005-29**

**December 2005, Prague**

# A Generic XML-Based Format for Structured Linguistic Annotation and Its Application to the Prague Dependency Treebank 2.0

by Petr Pajas and Jan Štěpánek

Published 2005

## Abstract

In the first part of this technical report we describe our approach to design a new data format, based on XML (Extensible Markup Language) and aimed to provide a better and unifying alternative to various legacy data formats used in various areas of corpus linguistics and specifically in the field of structured annotation.

We introduce the first version of the format, called Prague Markup Language (PML). This version has already been employed as the main data format for the upcoming Prague Dependency Treebank 2.0 (PDT).

Finally we outline our ideas and proposals for further improvement of PML, based on our current experience with using and processing data in PML format in the PDT 2.0 project.

The second part of the technical report contains the state-of-the-art specification of PML.

## Technická zpráva č. TR-2005-29

Technická zpráva projektu Integrace jazykových zdrojů za účelem extrakce informací z přirozených textů

Projekt Informační společnosti Grantové agentury Akademie věd ČR

Registrační číslo GA AV ČR: 1ET101120503

Interní kód MFF: 207-14 / 242083

# Table of Contents

# Part I. Towards a Generic Format for Linguistic Data

# Table of Contents

# Towards a Generic Format for Linguistic Data

Petr Pajas
Jan Štěpánek

## 1. Motivation

Due to the lack of a sufficiently acceptable and unifying data format for representing linguistic resources (either dictionaries or corpora and annotations), it became a common habit that each project created a format of its own. Such a format usually suited the needs of a particular project but was not reusable for other purposes. While this may prove inevitable in the global context, it should at least be striven for a solution within a single institution (or several cooperating institutions).

The same situation applied to the Prague Dependency Treebank (PDT). However, during the long period of its systematic development, various demands on data formats emerged and crystallized. Several efforts to deal with this problem led us to initiate a research that would aim to establish a standard approach to data representation for future projects in this area.

Among the requests most often repeated were, to name just a few, the following:

- Stand-off annotation principles: Each layer of the linguistic annotation should be cleanly separated from the other annotation layers as well as from the original data. This allows for making changes only to a particular layer without affecting the other parts of the annotation and data.
- Cross-referencing and linking: Both links to external document and data resources and links within a document should be represented coherently. Diverse flexible types of external links are required by the stand-off approach. Supposed that most data resources (data, tag-sets, and dictionaries) use the same principles, they can be more tightly interconnected.
- Linearity and structure: The data format ought to be able to capture both linear and structure types of annotation. Linear type includes e.g. word and sentence order (in case of written text) or temporality (in case of speech data). As for the structural annotation, our primary concern is to allow capturing tree-like structures in a way that mirrors their logical nesting.
- Structured attributes: The representation should allow for associating the annotated units with complex and descriptive data structures, similar to feature-structures.
- Alternatives: The vague nature of the language often leads to more than one linguistic interpretation and hence to alternative annotations. This phenomenon occurs on many levels, from atomic values to compound parts of the annotation, and should be treated in a unified manner.
- Human-readability: The data format should be human-readable. This is very useful not only in the first phases of the annotation process, when the tools are not yet mature enough to

reflect all evolving aspects of the annotation, but also later, especially for emergency situations when e.g. an unexpected data corruption occur that breaks the tools and can only be repaired manually. It also helps the programmers while creating and debugging new tools.

- Extensibility: The format should be extensible to allow new data types, link types, and similar properties to be added. The same should apply to all specific annotation formats derived from the general one, so that one could incrementally extend the vocabulary with markup for additional information.

- XML based: Employing a commonly used generic markup language as the underlying data representation makes achieving the above mentioned goals much easier. XML (Extensible Markup Language, [XML]) is such a format that is widely deployed and offers many tools and libraries for various programming languages already. This also means, that existing validation tools and schema languages for XML can be applied on the new format, too.

After doing some background research and surveying several existing data formats we have concluded that none of them satisfies all our requirements:

Text Encoding Initiative

The Text Encoding Initiative (TEI) (see [TEI]) provides guidelines for representing a variety of literary and linguistic texts for online research, teaching, and preservation. The TEI format is either SGML or XML based. The XML-based format is very rich and among other provides means for encoding linguistic annotation, both morphologic and phrase-structure oriented, as well as some generic markup for graphs, networks, trees, feature-structures, and links. Capturing e.g. dependency trees in TEI would, however, require a great amount of additional work resulting in an incomprehensible and to a certain extent artificial solution. TEI XML makes extensive use of entities, an almost obsoleted feature of XML, that originates in SGML. TEI lacks explicit support for stand-off annotation style.

Annotation Graphs

Annotation Graphs (AG) are a formal framework for representing linguistic annotations of time series data (see e.g. [AG]). At the time of our survey, syntactic annotation was not directly supported by AG, although AG could be in theory abused for the purpose of representing almost arbitrary structured data. The primary strength of AG lies in representing annotation of a temporal (time-line based) sequence of events.

ISO/TC 37/SC 4

The objective of ISO/TC 37/SC 4 [TC37SC4] is to provide principles and methods for creating, coding, processing and managing language resources, such as written corpora, lexical corpora, speech corpora, dictionary compiling and classification schemes. This initiative is very new, originating in 2003, and by the time of our survey it had not produced any results, although some basic ideas can be captured from the talk called "Representation, data format, standards" presented in [DiaBruck] .

Various single-purpose formats

Single-purpose formats such as Penn-treebank [PENN], NEGRA Corpus [NEGRA], Tiger Corpus [TIGER], and PDT 1.0 formats (CSTS, FS) [PDT1] are all too limited in order to be employed in other projects than those of their origin.

Our survey covered also other resources such as the technology used in the MATE Workbench [MATE], AG-based project ATLAS [ATLAS], as well as various papers related to the topic of our survey, such as [Dipper], proposing some similar ideas to those expressed in our list of requirements.

One of the main obstacles in most of the surveyed cases is the requirement for the ability to adequately capture dependency-tree like annotation and annotations beyond morpho-syntactic analysis in general.

The unsatisfactory results of our survey and the necessity to have some data format for the upcoming release of PDT 2.0 [PDT2] had led us to the decision to develop a new data format with the ambition to replace the previous ones, satisfying the maximal amount of the generic requirements, that would be general and flexible enough so that it need not be replaced by another one as soon as some random demand or a novel annotation area turn up.

# 2. Prague Markup Language

PML (“The Prague Markup Language”), as released in PDT 2.0, is our first step towards the new generic XML based data format. As such, it is already capable of representing rich linguistic annotation of texts, such as morphological tagging, dependency trees, etc. in adequate and straightforward way, as demonstrated on PDT 2.0 data.

A formal description of PML and its various technical aspects are given in Part II, “State-of-the-art specification of the PML format”.

In PML, individual layers of annotation can be stacked one over another in a stand-off fashion and linked from in a consistent way. Each layer of annotation is described in a *PML schema* file, which can be imagined as a formalization of an abstract annotation scheme for the particular layer of annotation. In brief, the PML schema file describes which elements occur on the layer, how they are nested and structured, of which types the values occurring in them are, and what role they play in the annotation scheme. This *role* information can also be used by applications such as TrEd (see [TRED]) to determine an adequate way to present a PML instance to the user. Based on a PML schema, it is possible to generate various validation schemas, such as RelaxNG[1], hence formal consistency of instances of the PML schema can be verified using conventional XML-oriented tools.

PML schemas are sometimes referred to as *applications of PML*, whereas individual XML documents conforming to a particular PML schema are referred to as *instances* of that schema or as *PML instances* in general.

The annotation is expressed by means of XML elements and attributes, named and nested according to the associated PML schema. XML elements of a PML instance occupy a dedicated namespace `http://ufal.mff.cuni.cz/pdt/pml/`.

PML format offers unified representations for the most common annotation constructs, such as

[1] http://www.relaxng.org/

attribute-value structures
: i.e. structures consisting of attribute-value pairs. To avoid confusion with XML attributes, we usually refer to attributes of an attribute-value structure as *members*.

lists
: allowing several values of the same type to be aggregated in either an ordered sequence or an unordered list.

sequences
: representing sequences of values of different types and also providing rudimentary support for XML-like mixed content.

alternatives
: used for aggregating alternative annotations, ambiguity, etc.

links
: providing a unified method for cross-referencing within a PML instance and linking both among various PML instances (which includes links between layers of annotation) and to other external resources (in the present revision, these resources have to be XML-based).

As already briefly mentioned, PML introduces a concept of so called *PML-roles*, which is orthogonal to the concept of data typing. The information provided by PML roles identifies a construct as a bearer of additional higher-level property of the annotation, such as being a node of a dependency tree, being a unique identifier, etc. (see Section 4, “PML roles” in Part II, “State-of-the-art specification of the PML format”).

Every PML instance starts with a header where a PML schema can be associated with the instance and where all external resources, which the instance points to, are listed, together with additional information necessary for correct link resolving. The rest of the instance is dedicated to the annotation itself.

# 3. Application of PML to the PDT 2.0

We shall now briefly summarize how PML is applied to the particular case of the PDT 2.0. Further details, including actual PML schemas, can be found in the document [PDTMarkup].

PDT 2.0 contains annotation divided into up to four layers stacked one upon another, namely the *word layer* (w-layer), *morphological layer* (m-layer), *analytical layer* (a-layer), and *tectogrammatical layer* (t-layer). Each of the layers defines its own PML schema.

The word layer segments the text into documents and paragraphs, each of which consists of a flat sequence of tokens. Each document, paragraph, and token on the w-layer has a unique identifier and also a pointer to the original source of the data, which in the particular case of PDT 2.0 is a SGML based format used by the Institute of the Czech National Corpus[2].

The morphological layer provides morphological annotation of the w-layer by means of a sequence of annotations pertaining to morphological forms abstracted from tokens of the w-layer

[2] http://ucnk.ff.cuni.cz/

as well as annotation of sentence boundaries. A morphological form may relate to zero or more tokens of the w-layer. The case where a form relates to no w-layer token only occurs in cases where the sentence was originally misspelled and a certain token (usually a punctuation mark) otherwise required by grammar rules was completely skipped in the original text. The relation between the m-layer and w-layer is represented as links from a m-layer PML instance into the corresponding instance of the w-layer.

Annotation on the *analytical layer* in PDT 2.0 consists of a sequence of (analytical) trees pertaining to the sentences marked up on the m-layer. There is a 1:1 correspondence between nodes of the analytical tree and forms on the m-layer, represented by links from an a-layer instance into the corresponding m-layer instance.

Annotation on the *tectogrammatical layer* consists of a sequence of (tectogrammatical) trees each of which pertains to a certain analytical tree. The mapping between analytical and tectogrammatical trees is 1:1. However, the mapping between nodes of a tectogrammatical tree and nodes of the corresponding analytical tree is in general N:M (with 0 possible both for N and M), and in some cases the later mapping even crosses tree boundaries (it never crosses a document/file boundary). Again, these relations are represented by links from t-layer instances into the corresponding a-layer instances.

Tectogrammatical and analytical trees are dependency trees, represented in PML commonly as nested attribute-value structures. In this representation, a node of a tree is realized as an attribute-value structure with PML-role `#NODE`. Each node has a dedicated member with a PML-role `#CHILDNODES`, which contains a list of child-nodes of the node. Because of the auxiliary character of root nodes of the dependency trees of PDT 2.0, the structure representing the technical root of the tree is of a different type then the rest of the nodes (i.e. has a different set of members).

The dependency trees both on the a-layer and t-layer are ordered trees, although the semantics for these two orderings is very different in both cases. The ordering of nodes of analytical trees is rather technical and simply mirrors the ordering of the underlying m-layer, while the ordering on the tectogrammatical trees is an integral part of the tectogrammatical annotation and has a strong linguistic interpretation based on communicative dynamism. The implementation of the ordering in PML is, however, common for both cases. Each node has a dedicated member with PML-role `#ORDER` and integer value type. The value of this member represents the position of a particular node in the total order of the tree it belongs to.

Other interesting features of PDT 2.0 annotation worth mentioning are the annotation of coreference, implemented as links between tectogrammatical nodes, and the annotation of quoted parts of the sentences, which aggregates tectogrammatical nodes into so called quotation sets, consisting of nodes pertaining to a part of the text in quotes. Moreover, the annotation distinguishes between several types of quoting, such as citation, direct speech, use of meta-language, etc. A tectogrammatical node may belong to zero or more quotation sets. In PML this is realized by means of “coloring” nodes that belong to a quotation set. Formally it is done as follows: Each node has a member `quot` which is a list of structures, each of which consists of two members, the first being an ID (“color”) identifying a quotation set and the second a value determining the type of the quotation. In this representation, a node belongs to a quotation set if and only if the ID of the quotation set is listed in the `quot` attribute of the node.

The solution used for annotation of quotation sets has the obvious advantage that it does not introduce any new annotation structure parallel to the tectogrammatical tree. On the other hand, it has some undeniable disadvantages, too, namely the following:

- quotation sets are not represented explicitly, as objects with given IDs. Instead, they are encoded implicitly, merely as a set of labels or references to virtual IDs scattered among nodes in a tree
- the information about the type of the quotation is repeated with each member of the quotation set, although it is of course constant over a quotation set.

As no other part of the tectogrammatical annotation relates to the annotation of the quotation-sets, introducing an extra annotation layer stacked over the t-layer would seem to be a cleaner solution in this case.

# 4. Improvements planned for the future revision of PML

PML indeed proved mature enough to be capable of capturing all aspects of annotation as complex as the four annotation layers of PDT 2.0. Nevertheless, during this first-time experience with applying PML to a real annotation, we have observed and collected several demands and suggestions for future improvements that we discuss in brief below, without any particular order of importance.

- XML data type: It seems useful to allow arbitrary XML data as a value type, especially in those cases where there is a standard XML vocabulary for representing a given type of information. For example, should some formal representation of mathematical expressions and formulae be part of the annotation, then using MathML[3] vocabulary and namespace would seem the most natural way.
- Improved typing: So far, typing in PML is static, meaning that an instance must strictly follow the typing dictated by a PML schema. PML so far offers no means to reverse the situation, e.g. declare a type as implied by the instance.
- Inheritance: In PML, complex data types are created from simpler types by composition, which, with the exception of element sequence, only permits fully specified types. Hence, for instance, the set of structure members and their value types must be specified without ambiguity. By providing some kind of type inheritance at least for the attribute-value structure type, we could allow one to declare abstract types from which specific types could be derived, so that values of all the derived types could be used equaly as instances of the abstract type. In case of structures, the type of an instance could be distinguished e.g. by some dedicated member with role `#TYPE`.
- Inclusions: There should be some means to include parts of one PML schema into another. This would not only improve extensibility and simplify adding customizations to existing

[3] http://www.w3.org/Math/

schemata, but also eliminate duplicated schema code, which is currently a necessity in cases where one layer of annotation allows for embedding some units from another layer (e.g. by process referred to as "knitting" in Part II, "State-of-the-art specification of the PML format").

- Overriding: Even better implementation of inclusions could allow the top-level PML schema to selectively override declarations of certain types within the included PML schema, again reducing the amount of duplication of declarations when deriving a new PML schema from an existing one.

- Should alternatives be allowed to have an ID? Consider an alternative of values of some complex type, such as structure. Although each structure in the alternative can be associated with a unique ID, PML does not allow the alternative itself to have a unique ID of its own. In fact, this is not such a limitation as it may seem, since one can always refer to an alternative by referring to each of its members. Most people who pointed out this limitation couldn't provide a reasonable semantic for distinguishing between a link to an alternative as a whole and its particular member. However, for the cases where such clear distinction in semantics can be given, we suggest wrapping the alternative into a structure with a unique ID and a member whose value are the alternatives.

- Versioning: One of the biggest faults of the first version of PML as used in PDT 2.0 is the lack of versioning. This affects both numbering of revisions of the PML specification (i.e. the schema language) as such as well as that of individual PML schemas. It is an imperative requirement for the next version of PML to introduce this.

- As PDT 2.0 data are not accompanied by much meta-data, recommendations for a uniform representation of such information have not been included in the first revision. As meta-data are a current topic in other areas as well, a suitable solution might be to use some generic framework, such as Dublin Core[4] or RDF[5].

- Although PDT 2.0 annotation is based on several annotation dictionaries such as morphology and PDT Valency Lexicon (ValLex), we have not so far researched the possibility to adequately encode these resources (nor any other dictionary-like data) in PML.

[4] http://dublincore.org
[5] http://www.w3.org/RDF/

# Part II. State-of-the-art specification of the PML format

## Table of Contents

# The Prague Markup Language

Petr Pajas, Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics

## 1. Introduction

The Prague Markup Language (PML) is a common basis of an open family of XML-based data formats for representing rich linguistic annotations of texts, such as morphological tagging, dependency trees, etc. PML is an on-going project in its early stage. This documentation reflects the current status of the PML development.

PML tries to identify common abstract data types and structures used in linguistic annotations of texts as well as in lexicons (especially those intended for machine use in NLP) and other types of linguistic data, and to define a unified, straightforward and coherent XML-based representation for values of these abstract types. PML also emphasizes the following aspects of linguistic annotation: the stand-off annotation methodology, possibility to stack layers of annotation one over another, and extensive cross-referencing. PML also tries to retain simplicity, so that PML instances (actual PML representation of data) could be processed with conventional XML-oriented tools.

Unlike, e.g. TEI XML, XHTML or DocBook, PML by it self is not a full XML vocabulary but rather a system for defining such vocabularies.

A fully specified XML vocabulary satisfying the requirements constituted in this document is called an *application of PML*. An Application of PML is formally defined using a specialized XML file called *PML schema*. PML schema provides one level of abstraction over standard XML-schema languages such as Relax NG[1] or W3C XML Schemas[2]. It defines an XML vocabulary and document structure by means of *PML data types* and *PML roles*. An XML document conforming to a PML schema is a PML *instance* of the schema. PML data types, described in detail in Section 2, “PML data types”, include atomic types (identifiers, strings, integers, enumerated types, id-references, etc.), and complex types, that are composed from abstract types such as attribute-value structures (AVS), lists, alternatives, and mixed-type sequences. We refer to a value of a complex type as a *construct*. The information provided by PML roles is orthogonal to data typing. It identifies a construct as a bearer of additional higher-level property of the annotation, such as being a node of a dependency tree, or being a unique identifier (see Section 4, “PML roles”).

Based on a PML schema of a particular application of PML, it is possible to automatically derive a corresponding Relax NG schema that conventional XML-oriented tools can use to validate actual PML *instances* (see Section 9, “Tools”).

All XML tags used in applications of PML belong to a dedicated XML namespace

```
http://ufal.mff.cuni.cz/pdt/pml/
```

[1] http://www.relaxng.org/
[2] http://www.w3.org/XML/Schema

We will refer to the above namespace as *PML namespace*. PML schema files use the following XML namespace referred to as *PML schema namespace*:

```
http://ufal.mff.cuni.cz/pdt/pml/schema/
```

Currently PML reserves three element names from the PML namespace for representation technical elements: `LM` (for bracketing list members), `AM` (for bracketing alternative members), and `head` (for a common PML instance header described in detail in Section 5, “Header of a PML instance”).

## 2. PML data types

The PML currently recognizes the following abstract data types from which complex data types are built by means of composition:

atomic type (cdata)
: Atomic values are literal strings. The exact content of an atomic value may be further specified as its format (see Section 3, “Atomic data formats”). In the XML, atomic values are (depending on the context) represented either as CDATA (i.e. text) content of an element or as an attribute value.

enumerated types
: An atomic-value type defined as an exhaustive list of possible values of that type.

structures
: A structure is a versatile PML abstract type. Sometimes it is called a feature-structure, attribute-value structure or AVS. To avoid confusion with XML attributes, we refer to attributes of a structure as *members*. A structure is similar in nature to a `struct` type in the C programming language. A structure is fully specified by names, types and optionally roles for each of its members. Different members of the structure must have distinct names. The structure is represented in XML by an element whose only content are attributes and/or sub-elements representing the members of the structure. An attribute or sub-element representing a member is named by the member and its content is the XML representation of the member's value. The order of members in the structure as represented in XML may be arbitrary. Whether a particular member is represented by an attribute or a sub-element is specified in the PML schema, however, only members with values of atomic types can be represented by attributes. Some structure members may in the PML schema be formally declared as required, in which case they must appear in the structure and its XML representation and must have non-empty content. All members not explicitly declared as required are optional.

lists
: PML offers unified representation of both ordered and unordered lists of constructs of the same type (the *list member type*). PML lists represent data similar in nature to arrays in various programming languages. An XML element representing a construct of a list type must as its only child-nodes have either zero or more XML elements named `LM` (“List Member”), each representing a construct of the list member type, or else (as a compact representation of singleton lists) its content must be of the list member type. List member type can not be a list, i.e. lists of lists are not allowed. Technically, the difference between

ordered and unordered lists is only in the declaration. Ordered lists may still contain repeated member (members with the same value). Applications are only required to preserve the ordering of ordered lists.

alternatives
: Similar to unordered lists but different in usage and semantics are alternatives. Alternatives can be used to represent data where usually one value of a certain type is used, but under some circumstances several alternative (or parallel) values are allowed. An XML element representing an alternative of constructs of a certain type (*alternative member type*) is either a representation of a construct of that type (in case of a single value, i.e. no actual alternative values) or has as its only child-nodes two or more XML elements named `AM` ("Alternative Member"), each of which represents a construct of the alternative member type. Alternative member type must not be an alternative, i.e. alternatives of alternatives are not allowed.

sequences
: Sequences are similar to ordered lists but do not require their member constructs to be of the same type. Each member of a sequence is represented by an XML element whose name is bound in the sequence definition with the type of the construct it bears and whose content represents the value. Although applications must preserve the order of elements in a sequence, the definition of a sequence construct in the PML schema implies no restrictions on what the order could possibly be. There may be zero or more elements of a given name in the sequence. The PML schema may declare some of the elements of the sequence as required, in which case at least one element of that name must occur in the sequence. Additional information may be attached to elements of a sequence by means of XML attributes (which must also be declared in the PML schema).

# 3. Atomic data formats

PML currently only defines the atomic data formats listed below. In the future, specification for more formats will be added and/or some generic mechanism for introducing user-defined atomic formats will be added.

ID
: An identifier string, i.e. a string satisfying the `name` production[3] of the XML specification.

PMLREF
: An atomic value which is either an identifier or a string consisting of two identifiers separated by the character `#`. Values of this format usually represent a reference (link), see Section 7, "References in PML".

nonNegativeInteger
: A non-negative integer value represented in decimal notation

any
: Arbitrary string of characters (used in all cases not covered by the formats above).

[3] http://www.w3.org/TR/2004/REC-xml-20040204/#NT-Name

# 4. PML roles

PML roles indicate a formal role that a given construct plays in the annotation schema. Roles are orthogonal to types, but usually are compatible only with certain types of constructs. Roles are primarily intended to be used by applications processing the data. So far the following roles have been specified:

#TREES
: Only applicable to a list or sequence constructs. This role identifies a construct whose member constructs represent dependency or constituency trees.

#NODE
: Only applicable to a structure or a sequence-member construct. This role identifies a node of a dependency or constituency tree.

#CHILDNODES
: Only applicable to a member of a list type in a structure with role `#NODE` or to a sequence in an element of role `#NODE`. This role identifies a construct representing a list of child-nodes of a node node in a dependency or constituency tree.

#ID
: Only applicable to an atomic construct, typically with format `ID`. A value with this role uniquely identifies a construct (an XML element, structure, sequence, etc.) in the PML instance. This means that all values with role `#ID` within a PML instance are distinct..

#KNIT
: This role indicates that the application may resolve the atomic value(s) as PML references and replace their content with copies of the referenced PML constructs. This role is only applicable to either:

  - a structure member of atomic type with PMLREF format
  - an sequence element of atomic type with PMLREF format
  - a list with atomic member type formated as PMLREF. The list must occur as a value of a sequence element or a structure member.

#ORDER
: This role identifies a structure member containing a non-negative integer value used for ordering nodes in an ordered tree.

#HIDE
: This role identifies a structure member whose non-zero non-empty value indicates that an application may hide the structure from the user.

# 5. Header of a PML instance

Every PML instance starts with the `header` element which must occur as the first sub-element of the document element. The header element has the following sub-elements:

`schema`
: Associates the instance with a PML schema file, indicating that the instance conforms to the associated schema. The filename or URL of the PML schema file is specified in the attribute `href`.

`references`
: This element contains zero or more `reffile` sub-elements, each of which maps a filename or URL (attribute `href`) of some external resource to an identifier (attribute `id`) used as aliases when referring to the resource from the instance (see Section 7, “References in PML”). If the external resource is an instance bound with the current instance as declared in the PML schema, then `reffile` must have also a third attribute, `name`, containing the name used in the tag `reference` in PML schema declaration of the bound instance. For every resource bound to the instance in the PML schema (using `reference` tag) there must be a corresponding `reffile`.

# 6. PML Schema File

In this section we describe the syntax of a PML schema file. We use the usual DTD-like expressions for describing content of individual elements, i.e.:

*`name`*
: lower-case literals denote names of XML elements

`PCDATA`
: denotes arbitrary text content

`EMPTY`
: denotes empty content

`(...)`
: brackets delimit groups of adjacent content

`?`
: indicates that the element or group whose specification immediately precedes is optional

`*`
: indicates that the element or group whose specification immediately can be repeated

`|`
: separates specifications of exclusively alternative content

`,`
: separates specifications of adjacent content

A formal definition of PML schema file syntax is available as a Relax NG schema, see Appendix II.A, *Relax NG for PML Schema*.

All elements of the PML schema file belong to the PML schema namespace. The following elements may occur in a PML schema:

`pml_schema`
: This is the root element of a PML schema file. It may have no attributes (except for the `xmlns` declaration of the PML-schema namespace).

  **Content:** `(description?, reference*, root, type*)`

`description`
: This element provides an optional short description of the PML schema.

  **Content:** `PCDATA`

`reference`
: This element declares, that each instance of the PML schema is bound with another PML instance (usually of a different PML schema) and provides a hint for an application on how to process the bound instance.

  ### Attributes

  `name`
  : a symbolic name for the bound instance. This name is used in the `reffile` element in the referring file's header to identify the bound instance (see Section 5, “Header of a PML instance”).

  `readas`
  : the value `trees` instructs the application to read the bound instance as a sequence of dependency or constituency trees; value `dom` instructs the application to read the bound instance using the generic Document object model.

`root`
: Defines the root element of a PML instance.

  **Content:** `(attribute*, (alt | list | choice | constant | structure | sequence | cdata | (element*, sequence) | element*))`

`type`
: Defines a named type. Named types are referred to from other elements using the attribute `type`. A named type may only be referred from contexts where the actual type represented by the named type is allowed. In other words, if an element in a PML schema refers to a named type, then the content of the named type definition must be also a valid content for the referring element.

### Attributes

`name`
: The name of the new named type (required)

`role`
: The PML role of constructs of the type (optional)

**Content:** `(attribute*, (alt | list | choice | constant | structure | sequence | cdata | (element*, sequence) | element*))`

`structure`
: Declares a complex type which is a structure with the specified members. Its content consists of one or more `member` elements defining members of the structure.

### Attributes

`name`
: An optional name of the type. This name is not used in the PML schema, but may be used by applications, e.g. when presenting constructs of the type to the user. (optional)

`role`
: The PML role of the constructs of the type (optional)

**Content:** `(member)+`

`member`
: Declares a member of a structure. The content of the element `member` defines the member's value type (unless a named-type is specified using the `type` attribute).

### Attributes

`name`
: Name of the member (required)

`required`
: value `1` declares the member as required, value `0` declares the member as optional (default is 0)

`role`
: PML role of the member's value (optional)

`as_attribute`
: value `1` declares that the member is in XML realized as an attribute of the element realizing the structure. In that case, the value type must be atomic. Value `0` declares, that the member is realized as an XML element whose content realizes the value construct. In the latter case case no restrictions are put on the value type (default is 0)

`type`
: declares, that the value type is the given named type (complementary to content)

**Content:** `(alt | list | choice | constant | structure | sequence | cdata)`

`list`
Defines a complex type as a list of constructs of a given type. The content defines the type of the list members (unless a named type is specified in the `type` attribute).

### Attributes

`ordered`
: value `1` declares an ordered list, value `0` declares an unordered list (required)

`type`
: declares that the constructs contained in the list are of a given named type (complementary to content)

`role`
: PML-role of constructs of the type - currently only roles `#KNIT` and `#CHILDNODES` may be used with lists (optional)

**Content:** `(alt | choice | constant | structure | sequence | cdata)`

`alt`
Defines a type which is an alternative of constructs of a given type. The content defines a type of the alternative members (unless a named type is specified in the `type` attribute).

### Attributes

`type`
: declares that the constructs contained in the list are of a given named type (complementary to content)

**Content:** `(list | choice | constant | structure | sequence | cdata)`

`choice`
Defines an enumerated type with a set of possible values specified in the `value` sub-elements.

**Content:** `(value)+`

`value`
The text content of this element is one of the values of an enumerated type.

**Content:** `PCDATA`

`cdata`
Defines an atomic type. Constructs of atomic types are represented in XML as text or attribute values. The atomic type is further specified using the `format` attribute which can have one of the values `ID`, `PMLREF`, `nonNegativeInteger`, `any`, described in Section 2, “PML data types”.

**Content:** `EMPTY`

`constant`
Defines an atomic type with a constant value specified in the content.

**Content:** `PCDATA`

`sequence`
Defines a type which is sequence of further specified XML elements.

## Attributes

`role`
PML role of constructs of the type (optional)

**Content:** `(element)+`

`element`
Defines a generic element construct by specifying its name, attributes and content type. Element constructs may only occur in sequences and other element constructs (including the top-level element if an instance defined in the PML schema by the `root` element).

One or more optional sub-elements `attribute` declare additional XML attributes that the element may have (these attributes must not collide with any attributes whose presence may be implied by the content construct). The rest of the allowed sub-elements specify the type of the content construct type which may be an atomic value, an alternative, a list, a structure, a sequence, or zero or more elements, optionally followed by a sequence, in which case all names of the elements preceding the sequence must be distinct from the names of the elements in the sequence (an application can recognize the start of a sequence in an element construct by encountering a sub-element listed in the sequence definition).

## Attributes

`name`
name of the XML element (required)

`role`
PML role of the construct (optional)

`required`
value `1` declares the element as required, i.e. at least one element of the name must occur in the sequence; value `0` declares the element as optional (default is 0)

`type`
: declares that the element's content is a construct of a given named type (complementary to content)

**Content:** `(attribute*, (alt | list | choice | constant | structure | sequence | cdata | (element*, sequence) | element*))`

`attribute`
: Defines an attribute of an element. The content defines the type of attribute's value.

### Attributes

`name`
: name of the attribute (required)

`required`
: value `1` declares the attribute as required, value `0` declares the attribute as optional (default is 0)

`role`
: defines a PML role of the attribute (optional)

`type`
: defines the type of the attribute value to be of a given named type. This named type must be an atomic construct (complementary to content)

**Content:** `(choice | cdata)`

# 7. References in PML

While it is likely that in the future PML will offer other kinds of references, such as XPointer, currently PML only defines syntax and semantics for simple ID-based references to PML structure, element or sequence constructs occurring either in the same or some other PML instance, and to XML elements of non-PML XML documents in general. Also, there is no syntax defined yet for references to non-XML resources or to constructs without an ID.

A reference to a construct occurring within the same PML instance is represented by the ID of the referred construct (see more specific definition below). A reference to an object occurring *outside* the PML instance is represented by a string consisting of a pair of identifiers separated by the # character. The first of the two identifiers is an ID associated in the header of the PML instance with the system file name or URL of the instance containing the referred object. The second of the identifiers is a unique ID of the construct (or element) within the PML (or XML) instance it occurs in.

If the referred construct is a structure, then its ID is the value of its member with the role `#ID`. If the referred construct is an element, then its ID is the value of its attribute with the role `#ID`. If the referred construct is an XML element in a non-PML XML document, then its ID is the value of its ID-attribute (e.g. either the attribute `xml:id` or some other attribute declared as ID in the document's DTD or schema).

## 8. Layers of annotation

PML references are suitable for stacking one layer of linguistic annotation upon another. For this purpose, the original text is usually transformed to a very simple PML instance that only adds the most essential features such as basic tokenization, identifiers of individual tokens, etc., providing the basis upon which further annotations could be stacked. If it is not possible or desirable to directly include tokens from the original text in such a base layer, then a suitable mechanism (currently not defined by PML) has to be employed in order to carry unambiguous references to the corresponding portions of the original text (regardless of the original format).

A specific PML schema is usually defined for each of the annotation layers. The relation between annotation layers is typically expressed on the instance level using PML references and on the PML schema level using the instance binding (PML schema element `reference`).

## 9. Tools

The XSLT stylesheet pml2rng.xsl[4] transforms a PML schema to the corresponding Relax NG schema that can be used for validating instances of the PML schema. The resulting Relax NG refers to a portion of Relax NG common to all PML applications which is stored in the file pml_common.rng[5].

There are many standard freely available tools that can be used to validate an XML document against a Relax NG, such as jing[6] or xmllint[7].

The Tree Editor TrEd[8] has built-in support for PML representation of dependency and constituency trees (see Section PMLBackend[9] in TrEd User's Manual[10] for details).

PML instances may also be processed using conventional XML-oriented tools without direct support for PML. One of them worth recommending is XSH[11], which is a versatile tool for XML processing.

[4] http://ufal.mff.cuni.cz/pdt2.0/tools/pml/pml2rng.xsl
[5] http://ufal.mff.cuni.cz/pdt2.0/tools/pml/pml_common.rng
[6] http://www.thaiopensource.com/relaxng/jing.html
[7] http://xmlsoft.org/
[8] http://ufal.mff.cuni.cz/~pajas/tred/index.html
[9] http://ufal.mff.cuni.cz/~pajas/tred/ar01s15.html#pmlbackend
[10] http://ufal.mff.cuni.cz/~pajas/tred/ar01-toc.html
[11] http://xsh.sourceforge.net

# II.A. Relax NG for PML Schema

In this appendix we provide a Relax NG schema for PML Schema files (it is a listing of the file `pml_schema.rng`[1]). Note, that this Relax NG schema is rather simplistic and that does not currently reflect all constraints implied on the syntax of the PML schema file expressed in this document. This especially includes constraints on applicability of certain roles on certain types of constructs, and also the requirement that a named type may only be referred from contexts where the actual type represented by the named type may occur.

```
<?xml version="1.0"?>
<!DOCTYPE grammar SYSTEM "/home/pajas/share/xml/relaxng.dtd">
<grammar xmlns="http://relaxng.org/ns/structure/1.0"
 xmlns:s="http://ufal.mff.cuni.cz/pdt/pml/schema/"
        xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <a:documentation>PML schema syntax</a:documentation>
  <start>
    <element name="s:pml_schema">
      <optional>
 <element name="s:description">
   <text/>
 </element>
      </optional>
      <zeroOrMore>
 <ref name="reference.element"/>
      </zeroOrMore>
      <element name="s:root">
 <attribute name="name"/>
 <ref name="element.content"/>
      </element>
      <zeroOrMore>
 <ref name="type.element"/>
      </zeroOrMore>
    </element>
  </start>

  <define name="reference.element">
    <element name="s:reference">
      <a:documentation>declare a bound instance and optinally provide
        a hint for applications on how to parse it</a:documentation>
      <attribute name="name"/>
      <optional>
 <attribute name="readas">
   <choice>
     <value>trees</value>
     <value>dom</value>
   </choice>
 </attribute>
      </optional>
    </element>
  </define>

  <define name="type.element">
    <element name="s:type">
      <a:documentation>a named complex type</a:documentation>
```

[1] http://ufal.mff.cuni.cz/pdt2.0/tools/pml/pml_schema.rng

```
     <attribute name="name">
         <data type="ID"/>
     </attribute>
     <optional>
<ref name="role.attribute"/>
     </optional>
     <ref name="element.content"/>
   </element>
 </define>

 <define name="type.attribute">
   <attribute name="type">
     <a:documentation>a reference to a named complex
       type</a:documentation>
     <data type="IDREF"/>
   </attribute>
 </define>

 <define name="attribute.element">
   <element name="s:attribute">
     <a:documentation>attribute declaration</a:documentation>
     <optional>
<attribute name="required">
  <choice>
    <value>0</value>
    <value>1</value>
  </choice>
</attribute>
     </optional>
     <attribute name="name"/>
     <optional>
<ref name="role.attribute"/>
     </optional>
     <choice>
<ref name="type.attribute"/>
<choice>
  <ref name="choice.element"/>
  <ref name="cdata.element"/>
</choice>
     </choice>
   </element>
 </define>

 <define name="role.attribute">
   <attribute name="role">
     <a:documentation>PML role of the value</a:documentation>
     <choice>
<value>#TREES</value>
<value>#NODE</value>
<value>#ORDER</value>
<value>#CHILDNODES</value>
<value>#ID</value>
<value>#KNIT</value>
<value>#HIDE</value>
     </choice>
   </attribute>
 </define>

 <define name="structure.element">
   <element name="s:structure">
```

```
     <a:documentation>a structure (AVS)</a:documentation>
     <optional>
<attribute name="name"/>
     </optional>
     <optional>
 <ref name="role.attribute"/>
     </optional>
     <oneOrMore>
<ref name="member.element"/>
     </oneOrMore>
   </element>
 </define>

 <define name="alt.element">
   <element name="s:alt">
     <a:documentation>an alternative of values of the same
       type</a:documentation>
     <choice>
<ref name="type.attribute"/>
<ref name="list.element"/>
<ref name="data.types"/>
     </choice>
   </element>
 </define>

 <define name="list.element">
   <element name="s:list">
     <a:documentation>a list of values of the same
       type</a:documentation>
     <attribute name="ordered">
<choice>
  <value>1</value>
  <value>0</value>
</choice>
     </attribute>
     <choice>
<group>
  <attribute name="role">
    <value>#KNIT</value>
  </attribute>
  <attribute name="type">
    <a:documentation>a reference to a named complex type
             for knitting</a:documentation>
    <data type="IDREF"/>
  </attribute>
  <ref name="cdata.element"/>
</group>
       <group>
         <optional>
    <ref name="role.attribute"/>
         </optional>
         <choice>
           <ref name="type.attribute"/>
        <ref name="alt.element"/>
     <ref name="data.types"/>
         </choice>
       </group>
     </choice>
   </element>
 </define>
 <define name="choice.element">
   <element name="s:choice">
     <a:documentation>enumerated type (atomic)</a:documentation>
     <oneOrMore>
<element name="s:value">
  <text/>
</element>
     </oneOrMore>
   </element>
 </define>

 <define name="cdata.element">
   <element name="s:cdata">
     <a:documentation>cdata type (atomic)</a:documentation>
     <attribute name="format">
<choice>
  <value>ID</value>
  <value>PMLREF</value>
  <value>nonNegativeInteger</value>
  <value>any</value>
</choice>
     </attribute>
   </element>
 </define>

 <define name="constant.element">
   <element name="s:constant">
     <a:documentation>a constant (atomic)</a:documentation>
     <text/>
   </element>
 </define>

 <define name="sequence.element">
   <element name="s:sequence">
     <a:documentation>a sequence of elements</a:documentation>
     <optional><ref name="role.attribute"/></optional>
     <oneOrMore>
<ref name="element.element"/>
     </oneOrMore>
   </element>
 </define>

 <define name="element.element">
   <element name="s:element">
     <a:documentation>an element of a sequence</a:documentation>
     <attribute name="name"/>
     <optional><ref name="role.attribute"/></optional>
     <optional><ref name="required.attribute"/></optional>
     <ref name="element.content"/>
   </element>
 </define>

 <define name="element.content">
   <zeroOrMore>
     <ref name="attribute.element"/>
   </zeroOrMore>
   <choice>
     <ref name="type.attribute"/>
     <ref name="alt.element"/>
     <ref name="list.element"/>
     <ref name="choice.element"/>
     <ref name="constant.element"/>
     <ref name="structure.element"/>
     <ref name="cdata.element"/>
     <group>
<zeroOrMore>
  <ref name="element.element"/>
</zeroOrMore>
<optional>
  <ref name="sequence.element"/>
</optional>
     </group>
   </choice>
 </define>

 <define name="required.attribute">
   <attribute name="required">
     <choice>
<value>0</value>
<value>1</value>
     </choice>
   </attribute>
 </define>

 <define name="member.element">
   <element name="s:member">
     <a:documentation>a member of a structure</a:documentation>
     <optional><ref name="required.attribute"/></optional>
     <optional>
<attribute name="as_attribute">
  <choice>
    <value>0</value>
    <value>1</value>
  </choice>
</attribute>
     </optional>
     <optional>
<ref name="role.attribute"/>
     </optional>
     <attribute name="name"/>

     <choice>
<ref name="type.attribute"/>
<ref name="alt.element"/>
<ref name="list.element"/>
<ref name="data.types"/>
     </choice>
   </element>
 </define>

 <define name="data.types">
   <choice>
     <ref name="choice.element"/>
     <ref name="constant.element"/>
     <ref name="structure.element"/>
     <ref name="sequence.element"/>
     <ref name="cdata.element"/>
   </choice>
 </define>
```

```
</grammar>
```

# II.B. Examples

In this appendix we provide some simple examples of PML usage. Rather than on practical applicability of the schemas that follow, we concentrate on demonstrating the features, definitions and representation of various constructs. We also show how PML references work and how annotation layers can be stacked one upon another.

## 1. Dependency trees

The following PML schema and instance show an application of PML to a very simple analytical dependency annotation of English sentences. In this example, the annotation consists of some meta data (annotator's name and a time stamp) and a list of trees. Each tree is represented by its root-node. Nodes are structures with two members bearing the node-labels (word form and its syntactical function) and two technical members (index of the node in the ordering of the tree - represented by attribute `ord`, and a list of child-nodes - represented by the element `governs`). Note, that if a list of child-nodes has only one member, then this single child-node may be directly represented by the `governs` element. This eliminates the need for an extra `LM` bracketing element. Note that PML doesn't actually distinguish between dependency trees and constituency trees, but since dependency trees are ordered trees and are not necessarily projective, we have to employ an extra member `ord` with PML-role `#ORDER` for the tree ordering. Because we do not want any linguistic complexity to distract the reader's attention from the technical aspects of how data are defined and represented in PML, we have chosen two shamelessly simple sentences.

## Example II.B.1: PML schema

```xml
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>Example of dependency tree annotation</description>
  <root name="annotation">
    <element name="meta">
      <structure>
        <member name="annotator"><cdata format="any"/></member>
        <member name="datetime"><cdata format="any"/></member>
      </structure>
    </element>
    <element name="trees" role="#TREES" required="1">
      <list type="node.type" ordered="1"/>
    </element>
  </root>
  <type name="node.type">
    <structure role="#NODE">
      <member name="ord" as_attribute="1" required="1" role="#ORDER">
        <cdata format="nonNegativeInteger"/>
      </member>
      <member name="func" type="func.type" required="1"/>
      <member name="form" required="1">
        <cdata format="any"/>
      </member>
      <member name="governs" role="#CHILDNODES" required="0">
        <list type="node.type" ordered="0"/>
      </member>
    </structure>
  </type>
  <type name="func.type">
    <choice>
      <value>Pred</value>
      <value>Subj</value>
      <value>Obj</value>
      <value>Attrib</value>
      <value>Adv</value>
    </choice>
  </type>
</pml_schema>
```

**Example II.B.2: Sample instance with annotation of the sentence: `John loves Mary. He told her this Friday.'**

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema href="example1_schema.xml"/>
  </head>
  <meta>
    <annotator>Jan Novak</annotator>
    <datetime>Sun May 1 18:56:55 2005</datetime>
  </meta>
  <trees>
    <LM ord="2">
      <func>Pred</func>
      <form>loves</form>
      <governs>
        <LM ord="1">
          <func>Subj</func>
          <form>John</form>
        </LM>
        <LM ord="3">
          <func>Obj</func>
          <form>Mary</form>
        </LM>
      </governs>
    </LM>
    <LM ord="2">
      <func>Pred</func>
      <form>told</form>
      <governs>
        <LM ord="1">
          <func>Subj</func>
          <form>He</form>
        </LM>
        <LM ord="3">
          <func>Obj</func>
          <form>her</form>
        </LM>
        <LM ord="5">
          <func>Adv</func>
          <form>Friday</form>
          <governs ord="4"> <!-- ditto -->
            <func>Attrib</func>
            <form>this</form>
          </governs>
        </LM>
      </governs>
    </LM>
  </trees>
</annotation>
```

## 2. Constituency trees

On two simple (and of course incomplete) examples we demonstrate how Penn-treebank-like constituency trees might be represented in PML. This situation differs from the dependency trees in two aspects: 1) with constituency trees we do not have to consider an external ordering of the nodes in the tree, 2) constituency trees usually distinguish between leafs (terminal nodes) and branching nodes (non-terminal nodes). In the first sample we deal with this by declaring two node types and using sequences instead of lists (since lists would require all members to be of the same type). In the second sample we provide a minimalist approach taking advantage of the fact that a non-terminal node has at most one non-terminal child, which in turn eliminates the need to represent leafs nodes as nodes at all. This, in combination with the possibility to reuse element names for the actual labels, provides a very compact XML notation very close to the labeled-bracket syntax of Penn Treebank.

## Example II.B.3: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>Example of constituency tree annotation</description>
  <root name="annotation">
    <element name="meta">
      <structure>
        <member name="annotator"><cdata format="any"/></member>
        <member name="datetime"><cdata format="any"/></member>
      </structure>
    </element>
    <sequence role="#TREES">
      <element name="nt" role="#NODE" type="nonterminal.type"/>
    </sequence>
  </root>
  <type name="nonterminal.type">
    <attribute name="pos">
      <choice>
        <value>S</value>
        <value>VP</value>
        <value>NP</value>
        <value>PP</value>
        <value>ADVP</value>
        <!-- etc. -->
      </choice>
    </attribute>
    <sequence role="#CHILDNODES">
      <element name="nt" role="#NODE" type="nonterminal.type"/>
      <element name="form" role="#NODE" type="terminal.type"/>
    </sequence>
  </type>
  <type name="terminal.type">
    <cdata format="any"/>
  </type>
</pml_schema>
```

### Example II.B.4: Sample instance with annotation of the sentence: `John loves Mary. He told her this Friday.'

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema href="example2_schema.xml"/>
  </head>
  <meta>
    <annotator>John Smith</annotator>
    <datetime>Sun May 1 18:56:55 2005</datetime>
  </meta>
  <nt pos="S">
    <nt pos="NP">
      <form>John</form>
    </nt>
    <nt pos="VP">
      <form>loves</form>
      <nt pos="NP">
        <form>Mary</form>
      </nt>
    </nt>
  </nt>
  <nt pos="S">
    <nt pos="NP">
      <form>He</form>
    </nt>
    <nt pos="VP">
      <form>told</form>
      <nt pos="NP"><form>her</form></nt>
      <nt pos="ADVP"><form>this Friday</form></nt>
    </nt>
  </nt>
</annotation>
```

For brevity, we will not repeat the `meta` element in the second sample.

### Example II.B.5: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>
    Example of very compact constituency tree annotation
  </description>
  <root name="annotation">
    <sequence role="#TREES">
      <element name="S" role="#NODE" type="nonterminal.type"/>
    </sequence>
  </root>
  <type name="nonterminal.type">
    <attribute name="form"><cdata format="any"/></attribute>
    <sequence role="#CHILDNODES">
      <element name="VP" role="#NODE" type="nonterminal.type"/>
      <element name="NP" role="#NODE" type="nonterminal.type"/>
      <element name="PP" role="#NODE" type="nonterminal.type"/>
      <element name="ADVP" role="#NODE" type="nonterminal.type"/>
      <!-- etc. -->
    </sequence>
  </type>
</pml_schema>
```

### Example II.B.6: Sample instance

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head><schema href="example3_schema.xml"/></head>
  <S>
    <NP form="John"/>
    <VP form="loves">
      <NP form="Mary"/>
    </VP>
  </S>
  <S>
    <NP form="He"/>
    <VP form="told">
      <NP form="her"/>
      <ADVP form="this Friday"/>
    </VP>
  </S>
</annotation>
```

Note that once the labels of non-terminals coincide with the names of elements representing nodes, we could apply further restrictions on the nesting directly in the PML schema. For example, it would be very easy to incorporate some grammar-like rules (such as that `ADVP` can only occur within `VP`, etc.).

## 3. Internal references

To demonstrate cross-referencing in a PML instance we define two simple PML schemas for representing arbitrary graph with both labeled nodes. In the first schema, we represent the graph by a list of its vertices and a list of its edges. With the second schema the same graph is represented by a list of structures for nodes consisting of a label and a a list of pointers to the nodes connected with the current node by an edge.

## Example II.B.7: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>An oriented graph</description>
  <root name="graph">
    <structure>
      <member name="verteces">
        <list ordered="0">
          <structure>
            <member name="id" as_attribute="1" required="1" role="#ID">
              <cdata format="ID"/>
            </member>
            <member name="label" required="1">
              <cdata format="any"/>
            </member>
          </structure>
        </list>
      </member>
      <member name="edges">
        <list ordered="0">
          <structure>
            <member name="from.rf" required="1" as_attribute="1">
              <cdata format="PMLREF"/>
            </member>
            <member name="to.rf" required="1" as_attribute="1">
              <cdata format="PMLREF"/>
            </member>
          </structure>
        </list>
      </member>
    </structure>
  </root>
</pml_schema>
```

## Example II.B.8: Sample instance

```
<?xml version="1.0"?>
<graph xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head><schema href="example4_schema.xml"/></head>
  <verteces>
    <LM id="v1"><label>A</label></LM>
    <LM id="v2"><label>B</label></LM>
    <LM id="v3"><label>A</label></LM>
    <LM id="v4"><label>C</label></LM>
    <LM id="v5"><label>D</label></LM>
  </verteces>
  <edges>
    <LM from.rf="v1" to.rf="v2"/>
    <LM from.rf="v1" to.rf="v3"/>
    <LM from.rf="v2" to.rf="v4"/>
    <LM from.rf="v3" to.rf="v4"/>
    <LM from.rf="v4" to.rf="v1"/>
  </edges>
</graph>
```

## Example II.B.9: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>An oriented graph</description>
  <root name="graph">
    <list ordered="0">
      <structure>
        <member name="id" as_attribute="1"
                required="1" role="#ID">
          <cdata format="ID"/>
        </member>
        <member name="label" required="1">
          <cdata format="any"/>
        </member>
        <member name="edges.rf">
          <list ordered="0">
            <cdata format="PMLREF"/>
          </list>
        </member>
      </structure>
    </list>
  </root>
</pml_schema>
```

## Example II.B.10: Sample instance

```
<?xml version="1.0"?>
<graph xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head><schema href="example5_schema.xml"/></head>
  <LM id="v1">
    <label>A</label>
    <edges.rf>
      <LM>v2</LM>
      <LM>v3</LM>
    </edges.rf>
  </LM>
  <LM id="v2">
    <label>B</label>
    <edges.rf>v4</edges.rf>
  </LM>
  <LM id="v3">
    <label>A</label>
    <edges.rf>v4</edges.rf>
  </LM>
  <LM id="v4">
    <label>C</label>
    <edges.rf>v1</edges.rf>
  </LM>
  <LM id="v5">
    <label>D</label>
  </LM>
</graph>
```

# 4. External references

In this example we define PML schemas for two annotation layers. The first layer represents the tokenized text with the sentence-boundary markup. The second layer is a constituency-tree annotation of the sentences on the first (lower) layer. This constituency annotation is similar to the samples Section 2, “Constituency trees”, but this time the terminals contain references to the tokenized text.

The following schema and instance show a tokenization layer.

## Example II.B.11: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>Example of tokenization layer</description>
  <root name="tokenization">
    <element name="sentences">
      <list ordered="1" type="sentence.type"/>
    </element>
  </root>
  <type name="sentence.type">
    <structure>
      <member name="id" role="#ID" required="1" as_attribute="1">
        <cdata format="ID"/>
      </member>
      <member name="tokens"> <!-- words (tokens) -->
        <sequence>
          <element name="w">
            <attribute name="id" role="#ID" required="1">
              <cdata format="ID"/>
            </attribute>
            <cdata format="any"/>
          </element>
        </sequence>
      </member>
    </structure>
  </type>
</pml_schema>
```

## Example II.B.12: Sample instance

```
<?xml version="1.0"?>
<tokenization xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head><schema href="example6_schema.xml"/></head>
  <sentences>
    <LM id="s1">
      <tokens>
        <w id="s1w1">John</w>
        <w id="s1w2">loves</w>
        <w id="s1w3">Mary</w>
        <w id="s1w4">.</w>
      </tokens>
    </LM>
    <LM id="s2">
      <tokens>
        <w id="s2w1">He</w>
        <w id="s2w2">told</w>
        <w id="s2w3">her</w>
        <w id="s2w4">this</w>
        <w id="s2w5">Friday</w>
        <w id="s2w6">.</w>
      </tokens>
    </LM>
  </sentences>
</tokenization>
```

The following schema and instance show an annotation layer stacked over the previously defined tokenization layer. The relation between units on these layers, represented by PML references from the annotation layer to the tokenization layer, may in general be N to M. The references to the tokenization layer have role `#KNIT`, which indicates that applications may replace the member `w.rf` containing the list of references with the corresponding object from the lower layer (i.e. the `w` element).

## Example II.B.13: PML schema

```
<?xml version="1.0"?>
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>
    Example of tree annotation over a tokenization layer
  </description>
  <reference name="tokenization" readas="dom"/>
  <root name="annotation">
    <sequence role="#TREES">
      <element name="S" role="#NODE">
        <attribute name="sentence.rf">
           <cdata format="PMLREF"/>
        </attribute>
        <list ordered="1" role="#CHILDNODES" type="node.type"/>
      </element>
    </sequence>
  </root>
  <type name="node.type">
    <structure role="#NODE">
      <member as_attribute="1" name="pos">
        <choice>
          <value>S</value>
          <value>VP</value>
          <value>NP</value>
          <value>PP</value>
          <value>ADVP</value>
          <!-- etc. -->
        </choice>
      </member>
      <member name="w.rf">
        <list ordered="0" role="#KNIT" type="w.type">
          <cdata format="PMLREF"/>
        </list>
      </member>
      <member name="constituents" role="#CHILDNODES">
        <list ordered="1" type="node.type"/>
      </member>
    </structure>
  </type>
  <type name="w.type">
    <element name="w">
      <attribute name="id" role="#ID" required="1">
        <cdata format="ID"/>
      </attribute>
      <cdata format="any"/>
    </element>
  </type>
</pml_schema>
```

## Example II.B.14: Sample instance

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema href="example7_schema.xml"/>
    <references>
      <reffile name="tokenization" id="t" href="example6.xml"/>
    </references>
  </head>
  <S sentence.rf="s1">
    <LM pos="NP"><w.rf>t#s1w1</w.rf></LM>
    <LM pos="VP">
      <w.rf>t#s1w2</w.rf>
      <constituents pos="NP">
        <w.rf>t#s1w3</w.rf>
      </constituents>
    </LM>
  </S>
  <S sentence.rf="s2">
    <LM pos="NP"><w.rf>t#s2w1</w.rf></LM>
    <LM pos="VP">
      <w.rf>t#s2w2</w.rf>
      <constituents>
        <LM pos="NP"><w.rf>t#s2w3</w.rf></LM>
        <LM pos="ADVP">
          <w.rf>
            <LM>t#s2w4</LM>
            <LM>t#s2w5</LM>
          </w.rf>
        </LM>
      </constituents>
    </LM>
  </S>
</annotation>
```

After knitting is applied on PML references in `w.rf`, the instance appears to the application as follows:

## Example II.B.15: Sample instance after knitting

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema href="example7_schema.xml"/>
    <references>
      <reffile name="tokenization" id="t" href="example6.xml"/>
    </references>
  </head>
  <S sentence.rf="s1">
    <LM pos="NP"><w id="s1w1">John</w></LM>
    <LM pos="VP">
      <w id="s1w2">loves</w>
      <constituents pos="NP">
        <w id="s1w3">Mary</w>
      </constituents>
    </LM>
  </S>
  <S sentence.rf="s2">
    <LM pos="NP"><w id="s2w1">He</w></LM>
    <LM pos="VP">
      <w id="s2w2">told</w>
      <constituents>
        <LM pos="NP"><w id="s2w3">her</w></LM>
        <LM pos="ADVP">
          <w>
            <LM id="s2w4">this</LM>
            <LM id="s2w5">Friday</LM>
          </w>
        </LM>
      </constituents>
    </LM>
  </S>
</annotation>
```

# References

[TEI] The TEI Consorcium, *TEI P5 - Guidelines for Electronic Text Encoding and Interchange*, C.M.Sperberg-McQueen and LouBurnard ed. (January 2005).
http://www.tei-c.org/P5/

[AG] Steven Bird and Mark Liberman, *A Formal Framework for Linguistic Annotation (revised version)* (2000).
http://arxiv.org/abs/cs/0010033

[Dipper] Stefanie Dipper *XML-based Stand-off Representation and Exploitation of Multi-Level Linguistic Annotation*, 2005, In Proceedings of Berliner XML Tage 2005 (BXML 2005), pp. 39-50, Berlin, Germany.
http://www.ling.uni-potsdam.de/~dipper/papers/xmltage05.pdf

[MATE] D. McKelvie, A. Isard, A. Mengel, M.B. Møller, M. Grosse, M. Klein, 2001. *The MATE Workbench - an annotation tool for XML coded speech corpora*, Speech Communication 33 (1-2), pp. 97-112. Special Issue Speech Annotation and Corpus Tools.
http://www.ltg.ed.ac.uk/~amyi/papers/speechcomm00.ps

[ATLAS] S. Bird, D. Day, J. Garofolo, J. Henderson, C.L. Laprun, 2000, *ATLAS: A Flexible and Extensible Architecture for Linguistic Annotation*, In Proceedings of the Second International Language Resources and Evaluation Conference, pp. 1699-1706. Paris, European Language Resources Association.
http://arxiv.org/pdf/cs/0007022

[RelaxNG] *RELAX NG Specification*, OASIS Committee Specification (3 December 2001). Definitive specification for RELAX NG using the XML syntax.
Project homepage: http://relaxng.org/

[XML] *Extensible Markup Language*, World Wide Web Consortium (W3C).
http://www.w3.org/XML/

[XSLT] *XSL Transformations (XSLT) Version 1.0*, W3C Recommendation (16 November 1999), JamesClark ed., World Wide Web Consortium (W3C).
http://www.w3.org/TR/xslt

[PDT1] Jan Hajič, Barbora Vidová-Hladká, Jarmila Panevová, Eva Hajičová, Petr Sgall, Petr Pajas, *The Prague Dependency Treebank 1.0 (Final Production Label)* (2001), Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Prague.
http://ufal.mff.cuni.cz/pdt1/

[PDT2] *The Prague Dependency Treebank, 2.0* beta version, Institute of Formal and Applied Linguistics, Faculty of Mathematics and PhysicsPrague (2005).
http://ufal.mff.cuni.cz/pdt2.0/

[PENN] *The Penn Treebank Project*, LINC Laboratory, Computer and Information Science Department, University of Pennsylvania published by Linguistic Data Consortium.
http://www.cis.upenn.edu/~treebank/

[NEGRA] *Negra Corpus Version 2*, Saarland University, Departmentof Computational Linguistics and Phonetics Saarbrücken, Germany.
http://www.coli.uni-saarland.de/projects/sfb378/negra-corpus/

[TIGER] *TIGER Corpus*.
Project homepage: http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERCorpus/

[PDTMarkup] *PDT 2.0 Annotation Markup Reference*.
http://ufal.mff.cuni.cz/pdt2.0/doc/data-formats/pml-markup/index.html

[TC37SC4] ISO/TC 37/SC 4.
Project homepage: http://www.tc37sc4.org/

[DiaBruck] *DiaBruck 2003 Tutorial: Best Practice in Empirically-based Dialogue Research*, David Traum, Laurent Romary, Michael Strube.
http://www.coli.uni-saarland.de/conf/diabruck/pages/tutorial.htm

[TRED] Tree Editor TrEd.
Project homepage: http://ufal.mff.cuni.cz/~pajas/tred