The Prague Markup Language (Version 1.1)

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

Revision History

Revision 1.0.0 5 Dec 2005

Initial revision for UFAL technical report no. TR-2005-29

Revision 1.0.1 4 Aug 2006

Added revision history; added missing list of allowed attributes to the specification of the PML schema element `root`

Revision 1.1.0 1 May 2006

This revision introduces schema language versioning, and several major changes concerning `sequence`, `element`, and `root` data types. In this revision, `element` is no more a separate data type, but only a syntactic construction of a sequence (similar to the `member` subelement of `structure`). A new data type `container` is introduced which replaces the previous data type function of `element`. Possible orderings of sequence elements can now be specified via a regular-grammar-based attribute `content_pattern`.

Newly introduced PML schema elements `import` and `derive` provide modularization support for PML schemas.

This revision also restricts the format `ID` to the `NCName` production[1] of Namespaces in XML[2] and introduces a set of new formats based on W3C XML Schema built-in simple types.

Revision 1.1.1 26 Jun 2006

Since this revision, the `root` can also be of a container type.

Revision 1.1.2 11 Jul 2006

Since this revision, PML schema may be embedded in the `schema` element of PML instance header (in which case `href` attribute is omitted).

Revision 1.1.3 20 Aug 2006

This revision clarifies that `type`, `root`, `structure`, `element`, `member`, and `attribute` names must match the `NCName` production[3] of Namespaces in XML[4] and that `root`, `member`, and `element` names must be different from `LM` and `AM`.

Additional constraints on the declarations with role `#KNIT` were added and minor edits for improved clarity were made.

Revision 1.1.4 19 July 2008

An `attribute` may also be of the type `constant`.

Revision 1.1.5 13 March 2010

The `role` attribute cannot be used with the `type` element. The use of the `type` attribute in connection with a `derive` element has been clarified. The use of `xml:id` as an attribute name explicitly allowed.

Table of Contents

[1] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[2] http://www.w3.org/TR/1999/REC-xml-names-19990114/

[3] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[4] http://www.w3.org/TR/1999/REC-xml-names-19990114/

Revision history markup: This is how a paragraph added between two latest major revisions looks like.

This is how a paragraph modified between two latest major revisions looks like.

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 the data) could be processed with conventional XML-oriented tools.

Unlike, e.g. TEI XML, XHTML or DocBook, PML by itself 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[5] or W3C XML Schemas[6]. It defines an XML

[5] http://www.relaxng.org/
[6] http://www.w3.org/XML/Schema

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, which 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 an 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 10.2, “Tools”).

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

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

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 the representation of the 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 described below. Specific types are are built from abstract types means of composition. For each abstract type, an example of a concrete declaration in the PML schema is given toghether with examples of the XML representation of the corresponding data in an instance conforming to that schema.

2.1. Character data 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 in XML either as a CDATA (i.e. text) content of an element or as an attribute value.

Example 1:

```
<type name="id.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <cdata format="anyURI"/>
</type>
```

Example 2:

```
<link>http://www.mff.cuni.cz</link>
```

2.2. Enumerated atomic type

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

Example 3:

```
<type name="pos.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <choice>
    <value>verb</value>
    <value>noun</value>
    <value>adjective</value>
    <value>adverb</value>
  </choice>
</type>
```

Example 4:

```
<pos>verb</pos>
```

2.3. Constant atomic type

This is like an enumerated type with just one possible value. Therefore, whenever e.g. an attribute or structure member is declared with content is of a constant type, its value must be equal to this constant. Moreover, unless the attribute or member is required, the constant value is assummed when the attribute or member is omitted.

Example 5:

```
<type name="nodetype.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <constant>terminal</constant>
</type>
```

Example 6:

```
<node/>
<node type="terminal"/>
```

2.4. 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 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.

Example 7:

```
<type name="morph-analysis.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <structure>
    <member name="id" role="#ID" required="1" as_attribute="1">
      <cdata format="ID"/>
    </member>
    <member name="pos" required="1" type="pos.type"/>
    <member name="case" type="case.type"/>
    <member name="gender">
      <!-- ... -->
    </member>
  </structure>
</type>
```

Example 8:

```
<some_element id="a1" xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <pos>noun</pos>
  <case><AM>1</AM><AM>4</AM></case>
</some_element>
```

2.5. 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 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.

Example 9:

```
<type name="morph-analyses.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <list type="morph-analysis.type" ordered="1"/>
</type>
```

Example 10:

```
<analyses xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <LM id="a1">
    <pos>noun</pos>
    <case><AM>1</AM><AM>4</AM></case>
  </LM>
  <LM id="a2">
    <pos>adjective</pos>
    <case>3</case>
    <gender>fem</gender>
  </LM>
  <LM id="a3">
    <pos>verb</pos>
  </LM>
</analyses>
```

Example 11: Example of an XML representation of a list with only one element

```
<analyses id="a1">
  <pos>noun</pos>
  <case><AM>1</AM><AM>4</AM></case>
</analyses>
```

2.6. 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.

Example 12:

```
<type name="case.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <alt>
    <container>
      <attribute name="probability"><cdata format="float"/></attribute>
      <cdata format="nonNegativeInteger"/>
    </container>
  </alt>
</type>
```

Example 13:

```
<case>
  <AM probability="0.7">1</AM>
  <AM probability="0.3">4</AM>
</case>
```

Example 14: Example of an XML representation of an alternative with only one member

```
<case probability="1.0">7</case>
```

2.7. 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. The order and number of occurrences of elements in a sequence may be specified by a regular expression or left unrestricted.

Example 15:

```
<type name="chapter-content.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <sequence content_pattern="title, para*, section*, see-also?">
    <element name="title"><cdata format="any"/></element>
    <element name="para" type="para.type"/>
    <element name="section" type="section.type"/>
  </sequence>
</type>
```

Example 16:

```
<chapter>
  <title>Introduction</title>
  <para>....</para>
  <para>....</para>
  <section>...</section>
  <section>...</section>
  <section>...</section>
  <see-also>...</see-also>
</chapter>
```

2.8. Containers

Containers are similar but simpler to structures and can be used to annotate a piece of data by a set of attribute-value pairs with text-only values. They are represented in XML by an element whose content is the data and XML attributes are the annotation. The content of a container can be of any type except for a container and structure.

Important

Because of the compact representation of singleton lists and alternatives, a special care should be taken when using containers with content whose type is a list or alternative of containers or structures in order to avoid possible collisions between names of the attributes of the container and attributes or members rendered as attribute of the contained (singleton) container or structure. (This problem also applies to type derivation and also to inheritance which is to appear in a future revision of this specification). To avoid such problems, applications serializing PML data to XML are allowed to surround singleton list or alternative members within a container by `LM` or `AM` tags respectively, and they must do so if a name collision is apparent from the PML schema.

Example 17:

```
<type name="chapter.type" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <container type="chapter-content.type">
    <attribute name="id" role="#ID" required="1">
      <cdata format="ID"/>
    </attribute>
    <attribute name="autor.rf" >
      <cdata format="PMLREF"/>
    </attribute>
  </container>
</type>
```

Example 18:

```
<chapter id="ch1" author.rf="authors#johnsmith">
  <title>Introduction</title>
  <para>....</para>
  <!-- ...etc... -->
</chapter>
```

3. Atomic data formats

PML currently recognizes the follwoing atomic data formats: In the future, specification for more formats will be added and/or some generic mechanism for introducing user-defined atomic formats will be added.

`any`

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

`ID`

An identifier string, i.e. a string satisfying the `NCName` production[7] of the W3C specification Namespaces in XML[8]. Note in particular that the specification explicitly forbids a colon (`:`) to occur within an identifier.

Example: `ab`, `doc1.para2`, and `_d3p9_34-a2`, are all valid identifiers,

(whereas `-ab`, `234a`, and `a:x34` are all *invalid*).

[7] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[8] http://www.w3.org/TR/1999/REC-xml-names-19990114/

`PMLREF`

An atomic value which either is of the `ID` format described above, or consists of two substrings of the format `ID` delimited by the character `#`. Values of this format usually represent a reference (link), see Section 9, “References in PML”.

Example: `doc1#chap2-para3` or `doc1`.

Formats borrowed from the W3C XML Schema specification:

PML further recognizes the following selected XML Schema[9] built-in simple types[10] as PML `cdata` formats (each format is specified to cover the lexical space of the corresponding simple type in the XML Schema specification without consraining facets): string[11] , normalizedString[12] , token[13] , base64Binary[14] , hexBinary[15] , integer[16] , positiveInteger[17] , negativeInteger[18] , nonNegativeInteger[19] , nonPositiveInteger[20] , long[21] , unsignedLong[22] , int[23] , unsignedInt[24] , short[25] , unsignedShort[26] , byte[27] , unsignedByte[28] , decimal[29] , float[30] , double[31] , boolean[32] , duration[33] , dateTime[34] , date[35] , time[36] , gYear[37] , gYear-

[9] http://www.w3.org/TR/xmlschema-0/
[10] http://www.w3.org/TR/xmlschema-0/#CreatDt
[11] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#string
[12] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#normalizedString
[13] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#token
[14] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#base64Binary
[15] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#hexBinary
[16] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#integer
[17] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#positiveInteger
[18] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#negativeInteger
[19] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#nonNegativeInteger
[20] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#nonPositiveInteger
[21] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#long
[22] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#unsignedLong
[23] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#int
[24] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#unsignedInt
[25] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#short
[26] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#unsignedShort
[27] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#byte
[28] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#unsignedByte
[29] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#decimal
[30] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#float
[31] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#double
[32] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#boolean
[33] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#duration
[34] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#dateTime
[35] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#date
[36] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#time
[37] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#gYear

Month[38] , gMonth[39] , gMonthDay[40] , gDay[41] , Name[42] , NCName[43] , anyURI[44] , language[45] , IDREF[46] , IDREFS[47] , NMTOKEN[48] , NMTOKENS[49]

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 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 structure with the role `#NODE` or a list or sequence content type of a container with the role `#NODE`. This role identifies a construct representing a list of child nodes of a node in a dependency or constituency tree.

#ID
: Only applicable to an atomic construct, typically with the format `ID`. A value with this role uniquely identifies a construct (a structure, sequence, container, etc.) in the PML instance. This means that all values with the 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 referenced PML constructs. In case of an in-memory representation, the application may even arrange the data structures so that the PML reference is replaced by a direct pointer to or a shared copy of the corresponding data structure in the referenced PML construct in a way that allows accessing the referenced data structure as if it were part of the referring PML instance and so that change to it in any of the instances immediately causes the same change to be visible to the other.

This role is only applicable to either:

[38] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#gYearMonth
[39] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#gMonth
[40] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#gMonthDay
[41] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#gDay
[42] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#Name
[43] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#NCName
[44] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#anyURI
[45] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#language
[46] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#IDREF
[47] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#IDREFS
[48] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#NMTOKEN
[49] http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html#NMTOKENS

- a structure member of `cdata` type with the PMLREF format
- a sequence element of `cdata` type with the PMLREF format
- a list of `cdata` members with the PMLREF format. The list must occur in a container or as 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 associated PML schema may be either *external*, in which case its filename or URL is specified in the attribute `href`, or *internal*, in which case there is no `href` attribute and the `schema` element contains a single subelement `pml_schema` from PML schema namespace containing the definition of the associated PML schema as specified in Section 6, “PML schema file”.

`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 9, “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 the 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, the syntax of a PML schema file is specified. We describe the content of individual PML schema elements by formal patterns similar to the grammar used in DTD for element-content model specification:

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 the PML schema file syntax is available as a Relax NG schema, see Appendix 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). It consists of an optional `description`, declarations of common instance `references`, Schema modularization instructions `import` and `derive`, a `root` declaration and zero or more declarations of named types (`type`).

PML schemas which contain no `import` and `derive` instructions are called *simplified PML schemas*. The section Section 7, “Processing modular PML schemas” describes how `import` and `derive` instructions are to be processed in order to obtain an equivalent simplified PML schema.

Attributes

`version`

Version of the PML specification the schema conforms to, currently `1.1`

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

This optional element is used to assign a revision number to a particular version of the PML schema, see Section 8, “Numbering revisions of PML schemas”

Content: `PCDATA`

`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”). The name should match the `NCName` production[50] of Namespaces in XML[51]. (required)

`readas`

the value `pml` instructs the application to read the bound instance as PML (default); the value `trees` instructs the application to read the bound instance as a PML instance containing dependency or constituency trees; value `dom` instructs the application to read the bound instance as plain XML using the generic Document Object Model (DOM). (optional)

`import`

This element instructs an application processing the PML schema to load `root` and `type` declarations from an external PML schema file specified in the attribute `schema` to the current PML schema. The way in which the declarations are to be combined is described in Section 7, “Processing modular PML schemas”.

Attributes

`type`

Name of a specific type to import from the external PML schema file.

`schema`

A filename or URL of the imported external PML schema file. If the URL is relative, the URL of the current document is used as the base URL. An implementation may aditionally provide further means of resolving relative URLs, for example, a list of user-

[50] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[51] http://www.w3.org/TR/1999/REC-xml-names-19990114/

defined local paths in which it can search for schemas referred to by relative URLs if not found relative to the current document.

`revision`
: Constrains `revision` of the imported schema to the specific value. See Section 8, "Numbering revisions of PML schemas" for information on comparing revision numbers. If this attribute is present, then `minimal_revision` and `maximal_revision` attributes should be absent. (optional)

`minimal_revision`
: Constrains the `revision` of the imported schema to revision numbers larger or equal to the one specified. See Section 8, "Numbering revisions of PML schemas" for information on comparing revision numbers. If this attribute present, then `revision` attribute should be absent. (optional)

`maximal_revision`
: Constrains `revision` of the imported schema to revisions numbers smaller or equal to the one specified. See Section 8, "Numbering revisions of PML schemas" for information on comparing revision numbers. If this attribute present, then `revision` attribute should be absent. (optional)

Content: `EMPTY`

`derive`

This element instructs an application processing the PML schema to create a new type declaration by extending or modifying an existing *base declaration* specified by the `type` attribute. The newly created type declaration is called the *derived declaration*. The base declaration may either be one explicitely given by a `type` element, or a previously derived one. The base declaration must declare one of the following types: `structure`, `sequence`, `container`, or `choice`.

The element `derive` must contain exactly one of the following subelements: `structure`, `sequence`, `choice`, `container`, corresponding to the type of the base declaration. In the context of the `derive` element, the content and semantics of either of the above listed subelements differs from what is defined elsewhere in this specification in the following way:

- each `member`, `element`, `attribute`, or `value` subelement either replaces a member, element, attribute, or value with the same name (or content in case of `value`) from the base declaration, if such a one exists, or adds a new member, element, attribute, or value declaration to the derived structure, sequence, container, or choice declaration (respectively).

  additionally, the subelement may contain zero or more `delete` instructions each specifying a member, element, attribute, or value of the base declaration to be omitted from the derived declaration.

See Section 7, "Processing modular PML schemas" for detailed instructions on processing the `derive` instruction.

Attributes

`type`
A name of the base type declaration (required).

`name`
A name for the derived type declaration. If not specified, the derived declaration replaces the base declaration (this feature should be used with care and, advisably, only for base declarations imported from external PML schemas). The name should match the `NCName` production[52] of Namespaces in XML[53]. (optional)

Content: `(structure | sequence | choice | container)`

`delete`
This instruction can only occur in a `structure`, `sequence`, `choice`, or `container` subelement of a `derive` element (and is therefore not included in the specifications of the content of these individual elements).

The content is a name of a `member`, `element`, `attribute`, or `value` of a base declaration to be omitted from the derived declaration; see `derive` and Section 7, "Processing modular PML schemas" for processing details.

Content: `#PCDATA`

`root`
Declaration of the root element of a PML instance. A PML schema which does not (after possible simplification) contain a `root` declaration does not by itself fully define an application of PML, but may be used as a source of an `import` instruction in another PML schema.

Attributes

`name`
The (local) name of the root element. The name should match the `NCName` production[54] of Namespaces in XML[55] and must be different from `LM` and `AM`. (required)

`type`
declares that the root-element's content is a construct of a given named type. This attribute is complementary to content, i.e. if this attribute is present, then `root` must be an empty element. The named type this attribute refers to must follow the content pattern specified below.

Moreover, if the root element's content is declared as a `container`, then the container content type may only be a `sequence`.

[52] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName
[53] http://www.w3.org/TR/1999/REC-xml-names-19990114/
[54] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName
[55] http://www.w3.org/TR/1999/REC-xml-names-19990114/

Content: `(structure | sequence | container)`

`type`

Declaration 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. The name should match the `NCName` production[56] of Namespaces in XML[57]. (required)

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

`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. The name should match the `NCName` production[58] of Namespaces in XML[59]. (optional)

`role`

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

Content: `(member)+`

`member`

Declares a member of a structure. The attribute `name` defines the name of the member. The type of the member's value is specified either by the content or using the `type` attribute. It is an error if a structure declaration contains two member declarations with the same name.

[56] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[57] http://www.w3.org/TR/1999/REC-xml-names-19990114/

[58] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName

[59] http://www.w3.org/TR/1999/REC-xml-names-19990114/

Attributes

`name`
: Name of the member. The name should match the `NCName` production[60] of Namespaces in XML[61] and must be different from `LM` and `AM`. (required)

`required`
: value `1` declares the member as required, value `0` declares the member as optional (default is 0). Required member must be non-empty.

`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 member is of a given named type. If this attribute is present then content of the element `member` should be empty, with the only exception when `role` is `#KNIT` in which case content should be a `cdata` definition with `PMLREF` format and `type` should be used to specify the type of the value after knitting).

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

`list`
: Defines a complex type as a list of constructs of a given type. The content or `type` defines the type of the list members.

Attributes

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

`type`
: declares that the member constructs are of a given named type. If this attribute is present then content of the `list` element should be empty, with the exception when `role` is `#KNIT` in which case content should be a `cdata` definition with `PMLREF` format and `type` should be used to specify the type of list members after knitting).

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

[60] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName
[61] http://www.w3.org/TR/1999/REC-xml-names-19990114/

Content: `(alt | choice | constant | structure | container |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 | container | 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 listed in Section 3, “Atomic data formats”.

Content: `EMPTY`

`constant`

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

Content: `PCDATA`

`sequence`

Defines a data type representing ordered sequences of zero or more constituents. Each constituent is either a string of text or a named element whose content data type is uniquely determined by the element's name. The declaration of a sequence

- specifies elements which can occur in the sequence, uniquely mapping element names to data types,
- indicates if text constituents are allowed to occur in the sequence (sequences permitting text constituents are called mixed-content sequences),

- and, optionally, provides a simple regular-expression-like pattern describing all admissible orderings of constituents (element and interleaved text) in the sequence

Two text constituents in a mixed-content sequences should never be adjacent, i.e. there must always be an element occurring between every two text constituents.

Attributes

`role`

PML role of constructs of the type (optional)

`content_pattern`

This attribute constraints the order in which the constituents are allowed to appear in the sequence by means of an expression called content pattern (very similar and conceptually equivalent to the grammar of the element content model declaration in DTD and also similar to the syntax used in the productions in this specification).

The content pattern is built on content particles (`cp`'s), which consist of constituent specifiers, choice lists of content particles, or sequence lists of content particles. The syntax of a content pattern is given by the following grammar production rules:

```
pattern       ::= ( choice | seq | cp )

cnst          ::= ( Element-name | '#TEXT' )

cp            ::= ( cnst | '(' choice ')' | '(' seq ')' ) quantifier?

quantifier    ::= ( '?' | '*' | '+' )?

choice        ::= WS? cp ( WS? '|' WS? cp )+ WS?

seq           ::= WS? cp ( WS? ',' WS? cp )* WS?

WS            ::=  (#x20 | #x9 | #xD | #xA)+
```

where `pattern` is the content pattern; a content particle (`cp`) represents one or more constituents, which may appear in the sequence on a position in which the content particle appears in the pattern; `Element-name` is a name of an element constituent (see element) and represents any element constituent with this name; the string `'#TEXT'` represents a text constituent; any of the content particles occurring in a `choice` group may appear in the sequence at a position in which the `choice` group appears in the pattern; content particles occurring in a `seq` group must each appear in the sequence in the order in which it is listed in the group; optional `quantifier` character following a content particle governs whether the content it represents may occur one or more (`+`), zero or more (`*`), or zero or one times (`?`). The absence of a quantifier means that the content specified by the content particle must appear exactly once; content particles, brackets, commas, etc. may be optionally separated by white-space (`WS`).

A sequence matches a content pattern if and only if there is a path through the content pattern, obeying the sequence, choice, and quantifier operators and matching each constituent in the sequence against a `cnst` production (`Element-name` or `'#TEXT'`). It is an error if the pattern allows two adjacent text constituents.

Note

For compatibility with some SGML based content model implementations, it is advisable (but not enforced) to avoid non-deterministic (1-ambiguous) content patterns such as `(a,b)*,a?` (see e.g. Appendix E Deterministic Content Models (Non-Normative)[62] to the XML 1.0 specification[63] and the pointers therein). In particular, a constituent should not match more than one occurrence of a `cnst` production in the content pattern.

Content: `text?, (element)+`

`text`

This element can be used at the beginning of the `sequence` element to indicate that the sequence is of mixed-content. In that case, every (maximal) contiguous character content (including white-space) occurring within the XML element representing the sequence is treated as a constituent of the sequence.

Content: EMPTY

`element`

Declares an element constituent of a sequence. The attribute `name` specifies its name and either the content or the `type` attribute defines the value type. It is an error if a `sequence` declaration contains two elements with the same name.

Attributes

`name`

name of the element. The name should match the `NCName` production[64] of Namespaces in XML[65] and must be different from `LM` and `AM`. (required)

`role`

PML role of the construct (optional)

`type`

declares that the value is of a given named type. If this attribute is present then `element` should be empty, with the only exception when `role` is `#KNIT` in which case content should be a `cdata` definition with `PMLREF` format and `type` should be used to specify the value type after knitting).

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

`container`

Declares a container type. A container consists of a content value accompanied by an annotation provided by a set of name-value pairs with atomic values called attributes. The declaration consists of zero or more `attribute` declarations, followed by the content type

[62] http://www.w3.org/TR/2004/REC-xml-20040204/#determinism
[63] http://www.w3.org/TR/2004/REC-xml-20040204/#determinism
[64] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName
[65] http://www.w3.org/TR/1999/REC-xml-names-19990114/

declaration. The content can be of any type except for container and structure. Containers with empty content (indicated by absence of a content type declaration) are permitted.

Attributes

`role`
PML role of the construct (optional)

`type`
declares that the content is a construct of a given named type. This attribute is complementary to the part in brackets in the content specification below. That is, if the `type` attribute is present, then `container` can only contain attribute declarations.

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

`attribute`
Defines an attribute of a `container`. The content defines the type of attribute's value.

Attributes

`name`
name of the attribute. The name must either match the `NCName` production[66] of Namespaces in XML[67] or be equal to '`xml:id`'. Note that the latter is particularly useful in combination with the role `#ID`, and allows applications that are not PML-aware to recognize the attribute as an identifier as described in the xml:id[68] specification. (required)

`required`
value `1` declares the attribute as required, i.e. one that must be present on its container; value `0` declares the attribute as optional (defaults to 0 - optional). Required attribute must be non-empty.

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

`type`
defines the type of the attribute value as a given named type. The named type must be atomic. (The `type` attribute is complementary to content.)

Content: `(choice | cdata | constant)`

[66] http://www.w3.org/TR/1999/REC-xml-names-19990114/#NT-NCName
[67] http://www.w3.org/TR/1999/REC-xml-names-19990114/
[68] www.w3.org/TR/xml-id/

7. Processing modular PML schemas

A *simplified PML schema* is one which does not contain any `import` and `derive`. Simplified PML schemas are thus self-contained.

This section describes how to process a PML schema containing `import` and `derive` instructions in order to obtain a simplified PML schema semantically equivalent to the original PML schema (we call two PML schemas semantically equivalent if they describe the same class of instances, mapping same data to same data types and identifying these types with the same PML roles).

We describe the process of simplification of a PML schema by means of modifications to the original PML schema, although a particular implementation might choose a different processing strategy. See Section 10.2, “Tools” for a pointer to a reference implementation of this process.

A PML schema processor must first process all `import` instructions in the order in which they appear in the PML schema and then process the `derive` instructions.

7.1. Processing `import` elements

We call *current schema* the PML schema containing the `import` element in turn, and *imported schema* the PML schema referred to by the attribute `schema` of the `import` element. The processing of the `import` instruction differs depending on the presence of the `type` attribute.

If the `type` attribute is present, the element is processed as follows:

- If the current schema contains a `type` declaration or a `derive` instruction whose attribute `name` equals to the value of the `type` attribute of the `import` element, then the processing of the `import` element stops and it is removed from the current schema (this includes cases when the type declaration was added to the current schema during processing of any preceding `import` elements).

- Otherwise, the imported schema is read from the file specified by an URL (absolute or relative to the location of the file containing the current schema) contained in the `schema` attribute of the element `import`.

- The imported schema is parsed and its `revision` number is mached against `revision` or `minimal_revision` and `maximal_revision` attributes of the `import` element (if any of them is present). More specifically, if `revision` attribute of `import` is present then the imported schema revision must be equal to it. If `minimal_revision` attribute of `import` is present then the imported schema revision must be greater or equal to it. If `maximal_revision` attribute of `import` is present then the imported schema revision must be less or equal to it. It is an error if the revision of the imported schema does not match these constraints. The details of revision numbering and comparison of revision numbers are given in Section 8, “Numbering revisions of PML schemas”.

- The imported schema is processed according to these instructions into a simplified PML schema. (It is an error if two or more PML schemas refer among themselves via `import` elements in a way that forms a cycle or if a PML schema refers via an import element to itself).

- A type declaration whose attribute name equals to the attribute type of the import element is located in the imported schema and copied to the current schema. It is an error if such a declaration cannot be found in the imported schema.
- Every named type referred to by a type attribute from any element occurring within the copied declaration is also copied from the imported schema to the current schema, unless a type declaration or a derive instruction with the same name already exists in the current schema. This step is repeated as long as there are copied type declarations referring to declarations in the imported schema for which there is no type declaration in the current schema with the same name (either a copied or an original one). In other words, after copying the first type declaration, other type declarations may be copied to the current schema so that all references to named types are satisfied.
- Finally, the import element is removed from the current schema.

If the attribute type of the import element is absent, the instruction is processed as follows:

- The imported schema is read from the file specified by the schema attribute of the import element, parsed and processed just as in the prior case.
- If the current schema does not contain root declaration and there is a root declaration in the imported schema, it is copied to the current schema.
- Every type declaration is copied from the imported schema to the current schema, unless there already is a type declaration with the same name in the current schema.
- The import element is removed from the current schema.

7.2. Processing derive elements

The derive instructions cannot be processed if the schema contains any non-processed import instructions.

The derive element has an attribute type referring to a named type declaration which will be called the *base declaration*. It is an error if the PML schema (after all preceding derive instructions have been processed) does not contain a corresponding base declaration, i.e. a declaration whose attribute name equals to the attribute type of the derive element.

If the derive element contains an attribute name specifying a *target declaration name*, the base declaration is copied to the PML schema as a new type declaration under the target declaration name. We refer to this copy as *target declaration*. It is an error if prior to creating the target declaration the PML schema already contained a named type declaration with the same name. If the name attribute of the derive element is absent, the target declaration is the base declaration.

The `derive` element and the target declaration must contain the same subelement, which is one of `structure`, `sequence`, `container`, or `choice`. We refer to the subelement of the `derive` element as *source* and to the subelement of the `type` element representing the target declaration as *target*.

For each attribute of the source with a non-empty value the corresponding attribute on the target is added or if already present, its value is changed to match the value on the source. If an attribute is present on both the source and target but its value on the source is empty, the attribute on the target is removed from the target element. All other attributes of the target are left unchanged.

If the source (and hence also the target) is a `structure` element, all `member` subelements of the source are copied into the target, unless the target `structure` already contains a `member` subelement with the same name, in which case the `member` subelement from the source replaces the corresponding subelement of the target `structure`. Then, for every `delete` subelement of the source, the `member` subelement of the target `structure` whose `name` attribute equals to the content of the `delete` subelement is removed from the target `structure`. It is an error if the source contains a `delete` for which there is no matching `member` subelement in the target `structure`.

The processing of source and target elements which are a `sequence`, a `container`, or a `choice` is defined analogously, replacing in the definition the words `structure` and `member` with `sequence` and `element`, `container` and `attribute`, or `choice` and `value`, respectively, except that for `choice`, every `delete` subelement of the source deletes the `value` subelement of the target `choice` that has the same content (there is no `name` attribute for `value` elements).

The `derive` element is removed from the PML schema after the source has been processed as described above.

8. Numbering revisions of PML schemas

For maintenance and modularization purposes it is advisible that every revision of a PML schema which adds or modifies type declarations is assigned a unique revision number. For this purpose, PML provides the element `revision` of the PML schema. A modular PML schema using the `import` instruction to import types from another PML schema may specify constraints on the revision number of the imported schema. This section defines the format for PML schema revision numbers and revision number comparison method (implying a total order on revision numbers). Consequent revisions of a single PML schema file must be numbered in a non-decreasing order.

Revision numbers should be strings constisting of one or more interleaved non-negative integer numbers and the character '`.`', starting and ending with a number.

For example, `12`, `0.2.223`, and `12.23.1.2.2` are all valid revision numbers, whereas `.3`, `-3`,`1.2.`, or `74..23` are not.

We now describe comparizon of two revision numbers. Let $R=r_1.r_2.....r_n$ and $S=s_1.s_2.....s_k$ be two revision numbers, where r_i (for $i=1,...,n$) and s_j (for $j=1,...,k$) are non-negative integers and let n be less or equal to k. Define $r_i=0$ for every $i>n$. Then $R=S$ if and only if

$r_i=s_i$ for every $i=1,\ldots,k$; $R<S$ if and only if $r_1<s_1$ or there is some $j<k$ such that $r_i=s_i$ for every $i=1,\ldots,j$ and $r_{j+1}=s_{j+1}$; otherwise $R>S$.

For example, `1.0.0=1`, `2.1.3.8<2.1.12.8`, and `2>1.9.8`.

9. 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 formatted according to `PMLREF` format, i.e., 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 a container, 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).

Example 19:

```
<document xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
   <head>
     <schema>
       <!-- embedded schema -->
       <s:pml_schema version="1.1" xmlns:s="http://ufal.mff.cuni.cz/pdt/pml/schema/">
         <s:root name="document">
           <s:structure>
      <s:member name="links">
               <s:list ordered="1">
          <s:cdata format="PMLREF"/>
               </s:list>
             </s:member>
      <s:member name="parts">
        <s:list ordered="0">
          <s:structure>
                   <s:member name="xml:id" role="#ID" as_attribute="1">
                     <s:cdata format="ID"/>
                   </s:member>
     <s:member name="title">...</s:membmer>
     <s:member name="body">...</s:membmer>
                 </s:container>
               </s:list>
             </s:member>
           </s:structure>
         </s:root>
       </s:pml_schema>
     </schema>
     <references>
       <!-- declare documents we will refer to -->
       <reffile id="p1" href="part_1.xml"/>
       <reffile id="p2" href="part_2.xml"/>
     </references>
   </head>
   <links>
     <LM>p1#d1002<LM/> <!-- reference to part with ID "d1002" in document "part_1.xml" -->
     <LM>p2#d2234<LM/> <!-- reference to part with ID "d2234" in document "part_2.xml" -->
     <LM>d3237<LM/> <!-- reference to part with ID "d3237" in this document -->
     <LM>d3664<LM/> <!-- reference to part with ID "d3664" in this document -->
   </links>
   <parts>
     <LM xml:id="d3237">
       <title>...</title>
       <body>...</body>
     </LM>
     <LM xml:id="d3664">
       <title>...</title>
       <body>...</body>
     </LM>
   </parts>
</document>
```

10. Final recommendations

10.1. 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`).

10.2. Tools

The XSLT stylesheet `pml2rng.xsl` distributed in a package called PML Toolkit[69] transforms a PML schema to the corresponding Relax NG schema that can be used for validating instances of the PML schema.

There are many standard freely available tools that can be used to validate an XML document against a Relax NG, such as jing[70] or xmllint[71].

The PML Toolkit[72] further contains a Perl script `pml_simplify` implementing the conversion from a modular PML schema to a simplified PML schema described in Section 7, “Processing modular PML schemas” (the script in fact wraps several implementations, one of which is based on XSLT 2.0).

Another tool from the PML Toolkit[73] called `pml_copy` performs copying, moving, compressing, and uncompressing of PML instances without breaking internal references between the copied files and other related PML instances.

Finally, the toolkit contains conversion tools between PML and other formats, in particular the CoNLL format, the formats of the Penn, Tiger, Sinica, Alpino, Arabic, Hydarabad, and Latin treebanks, etc.

The Tree Editor TrEd[74] has built-in support for PML representation of dependency and constituency trees (see Section PMLBackend[75] in TrEd User's Manual[76] for details).

[69] rewrite:pmltk
[70] http://www.thaiopensource.com/relaxng/jing.html
[71] http://xmlsoft.org/
[72] rewrite:pmltk
[73] rewrite:pmltk
[74] rewrite:tred
[75] rewrite:tred_manual#pmlbackend
[76] rewrite:tred_manual

PML instances may of course be processed using conventional XML-oriented tools without direct support for PML. One of many useful and freely available tools for XML processing is the XML Editing Shell[77].

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_1_1.rng` from the PML Toolkit[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. In particular, the Relax NG does not enforce constraints on applicability of roles nor the requirement that a named type may only be referred to in contexts where the actual type represented by the named type is permitted.

```
<?xml version="1.0"?>
<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"
        xmlns:sch="http://www.ascc.net/xml/schematron"
 datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <sch:ns prefix="s" uri="http://ufal.mff.cuni.cz/pdt/pml/schema/"/>
  <a:documentation>PML schema syntax (revision 1.1.3)</a:documentation>
  <start combine="choice">
    <ref name="pml_schema.element"/>
  </start>

  <define name="version">
    <value>1.1</value>
  </define>
  <define name="pml_schema.element">
    <element name="s:pml_schema">
      <attribute name="version">
 <ref name="version"/>
      </attribute>
      <optional>
 <element name="s:revision">
   <ref name="revisionno.type"/>
 </element>
      </optional>
      <optional>
 <element name="s:description">
   <text/>
 </element>
      </optional>
      <zeroOrMore>
 <ref name="reference.element"/>
      </zeroOrMore>
      <ref name="pml_schema.body"/>
    </element>
  </define>
  <define name="pml_schema.body">
    <zeroOrMore>
      <ref name="import.element"/>
```

[77] http://xsh.sourceforge.net
[1] rewrite:pmltk

```
   </zeroOrMore>
   <zeroOrMore>
     <ref name="derive.element"/>
   </zeroOrMore>
   <optional>
     <ref name="root.element"/>
   </optional>
   <zeroOrMore>
     <ref name="type.element"/>
   </zeroOrMore>
 </define>
 <define name="import.element.attributes">
   <optional>
     <attribute name="type">
<ref name="type_name.type"/>
     </attribute>
   </optional>
 </define>
 <define name="import.element">
   <element name="s:import">
     <a:documentation>
instruction to import type(s) and root element from another
       PML schema
     </a:documentation>
     <attribute name="schema">
<data type="anyURI"/>
     </attribute>
     <optional>
       <choice>
         <attribute name="revision"/>
         <group>
           <optional>
             <attribute name="minimal_revision">
 <ref name="revisionno.type"/>
      </attribute>
    </optional>
    <optional>
      <attribute name="maximal_revision">
 <ref name="revisionno.type"/>
      </attribute>
    </optional>
         </group>
       </choice>
     </optional>
     <ref name="import.element.attributes"/>
   </element>
 </define>

 <define name="derive.element">
   <element name="s:derive">
     <a:documentation>
instruction to derive new type from an existing one (including
       one that is imported)
     </a:documentation>
     <attribute name="type">
<data type="NCName"/>
     </attribute>
     <optional>
       <attribute name="name">
  <ref name="type_name.type"/>
```

```
       </attribute>
     </optional>
     <choice>
       <element name="s:structure">
         <optional>
           <attribute name="name">
      <data type="NCName"/>
    </attribute>
         </optional>
         <optional>
           <ref name="role.attribute"/>
         </optional>
         <zeroOrMore>
           <ref name="member.element"/>
         </zeroOrMore>
         <zeroOrMore>
           <ref name="delete.element"/>
         </zeroOrMore>
       </element>
      <element name="s:sequence">
         <optional><attribute name="content_pattern"/></optional>
         <optional><ref name="role.attribute"/></optional>
         <zeroOrMore>
           <ref name="element.element"/>
         </zeroOrMore>
         <zeroOrMore>
           <ref name="delete.element"/>
         </zeroOrMore>
       </element>
      <element name="s:container">
         <optional><ref name="role.attribute"/></optional>
  <optional><ref name="type.attribute"/></optional>
         <zeroOrMore>
           <ref name="attribute.element"/>
         </zeroOrMore>
         <zeroOrMore>
           <ref name="delete.element"/>
         </zeroOrMore>
       </element>
       <element name="s:choice">
         <zeroOrMore>
           <element name="s:value">
             <text/>
           </element>
         </zeroOrMore>
         <zeroOrMore>
           <ref name="delete.element"/>
         </zeroOrMore>
       </element>
     </choice>
   </element>
 </define>

<define name="delete.element">
   <element name="s:delete">
     <a:documentation>delete instruction in derive element (depending
       on the context, contains name of a structure member, sequence
       element, or choice value)</a:documentation>
     <text/>
   </element>
 </define>

 <define name="type.decl">
   <choice>
     <ref name="type.attribute"/>
     <ref name="any.type"/>
   </choice>
 </define>

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

 <define name="data.type">
   <choice>
     <ref name="choice.element"/>
     <ref name="constant.element"/>
     <ref name="structure.element"/>
     <ref name="sequence.element"/>
     <ref name="container.element"/>
     <ref name="cdata.element"/>
   </choice>
 </define>
 <define name="non-structure.type">
   <choice>
     <ref name="alt.element"/>
     <ref name="list.element"/>
     <ref name="choice.element"/>
     <ref name="constant.element"/>
     <ref name="sequence.element"/>
     <ref name="cdata.element"/>
   </choice>
 </define>

 <define name="role.attribute">
   <attribute name="role">
     <a:documentation>PML role of the value</a:documentation>
     <ref name="role.type"/>
   </attribute>
 </define>

 <define name="role.type">
   <choice datatypeLibrary="">
     <value type="string">#TREES</value>
     <value type="string">#NODE</value>
     <value type="string">#ORDER</value>
     <value type="string">#CHILDNODES</value>
     <value type="string">#ID</value>
     <value type="string">#HIDE</value>
   </choice>
 </define>

 <define name="root.element">
   <a:documentation>declaration of the root-element of a PML instance
   (except for the implicit obligatory &lt;head&gt;)</a:documentation>
   <element name="s:root">
```

```
     <attribute name="name">
<ref name="name.type"/>
     </attribute>
     <choice>
<ref name="type.attribute"/>
<ref name="sequence.element"/>
<ref name="structure.element"/>
<ref name="container.element"/>
     </choice>
   </element>
 </define>

 <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">
<data type="NCName"/>
     </attribute>
     <optional>
<attribute name="readas">
  <choice datatypeLibrary="">
    <value type="string">pml</value>
    <value type="string">trees</value>
    <value type="string">dom</value>
  </choice>
</attribute>
     </optional>
   </element>
 </define>

 <define name="type.element">
   <element name="s:type">
     <a:documentation>a named complex type</a:documentation>
     <attribute name="name">
<ref name="type_name.type"/>
     </attribute>
     <ref name="any.type"/>
   </element>
 </define>

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

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

```
<ref name="member.element"/>
     </oneOrMore>
     <sch:pattern name="duplicite members">
       <sch:rule context="s:member">
         <sch:report test="count(../s:member[@name = current()/@name])>1">
    Structure definition contains two or more member declarations with the same name.
  </sch:report>
       </sch:rule>
     </sch:pattern>
   </element>
 </define>

 <define name="member.element">
   <element name="s:member">
     <a:documentation>a member of a structure</a:documentation>
     <optional><ref name="required.attribute"/></optional>
     <choice>
<group>
  <attribute name="as_attribute">
    <value>1</value>
  </attribute>
  <attribute name="name">
    <choice>
      <data type="NCName"/>
      <value type="Name">xml:id</value>
    </choice>
  </attribute>
</group>
<group>
  <optional>
    <attribute name="as_attribute">
      <value>0</value>
    </attribute>
  </optional>
  <attribute name="name">
    <ref name="name.type"/>
  </attribute>
</group>
     </choice>
     <choice>
<group>
  <attribute name="role">
    <value datatypeLibrary="" type="string">#KNIT</value>
  </attribute>
  <attribute name="type">
    <a:documentation>a reference to a named complex type
    for knitting</a:documentation>
    <ref name="type_ref.type"/>
  </attribute>
  <ref name="cdata.element"/>
</group>
<group>
  <optional>
    <ref name="role.attribute"/>
  </optional>
  <ref name="type.decl"/>
</group>
     </choice>
   </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.type"/>
     </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">
<ref name="pml_schema_bool.type"/>
     </attribute>
     <choice>
<group>
  <attribute name="role">
    <value datatypeLibrary="" type="string">#KNIT</value>
  </attribute>
  <attribute name="type">
    <a:documentation>a reference to a named complex type
             for knitting</a:documentation>
    <ref name="type_ref.type"/>
  </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.type"/>
         </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 datatypeLibrary="">
  <value type="string">any</value>
  <value type="string">ID</value>
  <value type="string">PMLREF</value>
         <value type="string">string</value>
         <value type="string">normalizedString</value>
         <value type="string">token</value>
         <value type="string">base64Binary</value>
         <value type="string">hexBinary</value>
         <value type="string">integer</value>
         <value type="string">positiveInteger</value>
         <value type="string">negativeInteger</value>
         <value type="string">nonNegativeInteger</value>
         <value type="string">nonPositiveInteger</value>
         <value type="string">long</value>
         <value type="string">unsignedLong</value>
         <value type="string">int</value>
         <value type="string">unsignedInt</value>
         <value type="string">short</value>
         <value type="string">unsignedShort</value>
         <value type="string">byte</value>
         <value type="string">unsignedByte</value>
         <value type="string">decimal</value>
         <value type="string">float</value>
         <value type="string">double</value>
         <value type="string">boolean</value>
         <value type="string">duration</value>
         <value type="string">dateTime</value>
         <value type="string">date</value>
         <value type="string">time</value>
         <value type="string">gYear</value>
         <value type="string">gYearMonth</value>
         <value type="string">gMonth</value>
         <value type="string">gMonthDay</value>
         <value type="string">gDay</value>
         <value type="string">Name</value>
         <value type="string">NCName</value>
         <value type="string">anyURI</value>
         <value type="string">language</value>
         <value type="string">IDREF</value>
         <value type="string">IDREFS</value>
         <value type="string">NMTOKEN</value>
         <value type="string">NMTOKENS</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>
     <optional><attribute name="content_pattern"/></optional>
     <choice>
<oneOrMore>
  <ref name="element.element"/>
</oneOrMore>
<group>
  <interleave>
    <ref name="text.element"/>
    <oneOrMore>
      <ref name="element.element"/>
    </oneOrMore>
  </interleave>
</group>
     </choice>
     <sch:pattern name="duplicite elements">
<sch:rule context="s:element">
  <sch:report test="count(../s:element[@name = current()/@name])>1">
    Sequence definition contains two or more element declarations with the same name.
  </sch:report>
</sch:rule>
     </sch:pattern>
   </element>
 </define>

 <define name="text.element">
   <element name="s:text">
     <a:documentation>declare cdata for mixed-content
     sequence</a:documentation>
     <empty/>
   </element>
 </define>

 <define name="element.element">
   <element name="s:element">
     <a:documentation>an element of a sequence</a:documentation>
     <attribute name="name">
<ref name="name.type"/>
     </attribute>
     <choice>
<group>
  <attribute name="role">
    <value datatypeLibrary="" type="string">#KNIT</value>
  </attribute>
  <attribute name="type">
    <a:documentation>a reference to a named complex type
    for knitting</a:documentation>
    <ref name="type_ref.type"/>
  </attribute>
  <ref name="cdata.element"/>
</group>
<group>
  <optional>
    <ref name="role.attribute"/>
  </optional>
  <ref name="type.decl"/>
</group>
     </choice>
   </element>
 </define>
 <define name="container.element">
   <element name="s:container">
     <a:documentation>a simple container type</a:documentation>
     <optional>
<ref name="role.attribute"/>
     </optional>
     <zeroOrMore>
<ref name="attribute.element"/>
     </zeroOrMore>
     <optional>
<choice>
  <ref name="type.attribute"/>
  <ref name="non-structure.type"/>
</choice>
     </optional>
     <sch:pattern name="duplicite attributes">
<sch:rule context="s:attribute">
  <sch:report test="count(../s:attribute[@name = current()/@name])>1">
    Container definition two or more attribute declarations with the same name.
  </sch:report>
</sch:rule>
     </sch:pattern>
   </element>
 </define>

 <define name="attribute.element">
   <element name="s:attribute">
     <a:documentation>attribute of a container</a:documentation>
     <optional><ref name="required.attribute"/></optional>
     <attribute name="name">
<choice>
  <data type="NCName"/>
  <value type="Name">xml:id</value>
</choice>
     </attribute>
     <optional>
<ref name="role.attribute"/>
     </optional>
     <choice>
<ref name="type.attribute"/>
<choice>
  <ref name="constant.element"/>
  <ref name="choice.element"/>
  <ref name="cdata.element"/>
</choice>
     </choice>
   </element>
 </define>

 <define name="required.attribute">
   <attribute name="required">
     <ref name="pml_schema_bool.type"/>
   </attribute>
 </define>

 <define name="name.type">
   <data type="NCName">
     <except>
<choice datatypeLibrary="">
  <value type="string">LM</value>
```

```
   <value type="string">AM</value>
 </choice>
      </except>
    </data>
  </define>
  <define name="revisionno.type">
    <data type="NMTOKEN">
      <param name="pattern">[0-9]+(\.[0-9]+)*</param>
    </data>
  </define>

  <define name="pml_schema_bool.type">
    <choice datatypeLibrary="">
      <value type="string">0</value>
      <value type="string">1</value>
    </choice>
  </define>

  <!-- we can only combine this with pml_schema_1_2.rng if we use NCName here -->
  <!-- See also pml_schema_1_1_strict.rng, where these are ID/IDREF -->
  <define name="type_name.type">
    <data type="NCName"/>
  </define>
  <define name="type_ref.type">
    <data type="NCName"/>
  </define>

</grammar>
```

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 PML schema in Example B.1, “PML schema (`example1_schema.xml`)” and the instance instance in Example B.2, “Sample instance with the annotation of the sentence: `*John loves Mary. He told her this Friday.*' (`example1.xml`) ” 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 the 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 does not 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 B.1: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1"
  xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>Example of dependency tree annotation</description>
  <root name="annotation">
    <structure>
      <member name="meta" type="meta.type"/>
      <member name="trees" role="#TREES" required="1">
 <list type="node.type" ordered="1"/>
      </member>
    </structure>
  </root>
  <type name="meta.type">
    <structure>
      <member name="annotator"><cdata format="any"/></member>
      <member name="datetime"><cdata format="any"/></member>
    </structure>
  </type>
  <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 B.2: Sample instance with the 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) PML schema+instance examples, (Example B.3, “PML schema (`example2_schema.xml`)” + Example B.4, “Sample instance with annotation of the sentence: `*John loves Mary. He told her this Friday.*' (`example2.xml`)” and Example B.5, “PML schema (`example3_schema.xml`)” + Example B.6, “Sample instance (`example3.xml`)”) 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 leaf nodes (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 (as 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 terminal child, which in turn eliminates the need to represent leaf 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 B.3: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1"
  xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>Example of constituency tree annotation</description>
  <root name="annotation">
    <sequence role="#TREES" content_pattern="meta, nt+">
      <element name="meta" type="meta.type"/>
      <element name="nt" type="nonterminal.type"/>
    </sequence>
  </root>
  <type name="meta.type">
    <structure>
      <member name="annotator"><cdata format="any"/></member>
      <member name="datetime"><cdata format="any"/></member>
    </structure>
  </type>
  <type name="nonterminal.type">
    <container role="#NODE">
      <attribute name="label" type="label.type"/>
      <sequence role="#CHILDNODES">
 <element name="nt" type="nonterminal.type"/>
 <element name="form" type="terminal.type"/>
      </sequence>
    </container>
  </type>
  <type name="terminal.type">
    <container role="#NODE">
      <cdata format="any"/>
    </container>
  </type>
  <type name="label.type">
    <choice>
      <value>S</value>
      <value>VP</value>
      <value>NP</value>
      <value>PP</value>
      <value>ADVP</value>
      <!-- etc. -->
    </choice>
  </type>

</pml_schema>
```

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

```xml
<?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 label="S">
    <nt label="NP">
      <form>John</form>
    </nt>
    <nt label="VP">
      <form>loves</form>
      <nt label="NP">
        <form>Mary</form>
      </nt>
    </nt>
  </nt>
  <nt label="S">
    <nt label="NP">
      <form>He</form>
    </nt>
    <nt label="VP">
      <form>told</form>
      <nt label="NP"><form>her</form></nt>
      <nt label="ADVP"><form>this Friday</form></nt>
    </nt>
  </nt>
</annotation>
```

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

Example B.5: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1" 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" type="nonterminal.type"/>
    </sequence>
  </root>
  <type name="nonterminal.type">
    <container role="#NODE">
      <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>
    </container>
  </type>
</pml_schema>
```

Example 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 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 (Example B.7, “PML schema (`example4_schema.xml`)” and Example B.9, “PML schema (`example5_schema.xml`)”) 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 B.7: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1" 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 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 B.9: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <description>An oriented graph</description>
  <root name="graph">
    <structure>
      <member name="body" required="1">
 <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>
      </member>
    </structure>
  </root>
</pml_schema>
```

Example B.10: Sample instance

```
<?xml version="1.0"?>
<graph xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head><schema href="example5_schema.xml"/></head>
  <body>
    <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>
  </body>
</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 schema in Example B.11, “PML schema (`example6_schema.xml`)” and the instance in Example B.12, “Sample instance (`example6.xml`)” show a tokenization layer.

Example B.11: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <revision>0.2</revision>
  <description>Example of tokenization layer</description>
  <root name="tokenization">
    <structure>
      <member name="sentences">
 <list ordered="1" type="sentence.type"/>
      </member>
    </structure>
  </root>
  <type name="sentence.type">
    <structure>
      <member name="id" role="#ID" type="ID.type"
 required="1" as_attribute="1"/>
      <member name="tokens"> <!-- words (tokens) -->
        <sequence>
          <element name="w" type="w.type"/>
        </sequence>
      </member>
    </structure>
  </type>
  <type name="w.type">
    <container>
      <attribute name="id" role="#ID" type="ID.type"
 required="1"/>
      <cdata format="any"/>
    </container>
  </type>
  <type name="ID.type">
    <cdata format="ID"/>
  </type>
</pml_schema>
```

Example 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 schema and instance in Example B.13, “PML schema (`example7_schema.xml`)” and Example B.14, “Sample instance (`example7.xml`)” 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 B.13: PML schema

```
<?xml version="1.0"?>
<pml_schema version="1.1" 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">
 <container role="#NODE">
   <attribute name="sentence.rf">
     <cdata format="PMLREF"/>
   </attribute>
   <list ordered="1" role="#CHILDNODES" type="node.type"/>
 </container>
      </element>
    </sequence>
  </root>
  <type name="node.type">
    <structure role="#NODE">
      <member as_attribute="1" name="label">
        <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">
    <container>
      <attribute name="id" role="#ID" required="1">
        <cdata format="ID"/>
      </attribute>
      <cdata format="any"/>
    </container>
  </type>
</pml_schema>
```

Example 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 label="NP"><w.rf>t#s1w1</w.rf></LM>
    <LM label="VP">
      <w.rf>t#s1w2</w.rf>
      <constituents label="NP">
        <w.rf>t#s1w3</w.rf>
      </constituents>
    </LM>
  </S>
  <S sentence.rf="s2">
    <LM label="NP"><w.rf>t#s2w1</w.rf></LM>
    <LM label="VP">
      <w.rf>t#s2w2</w.rf>
      <constituents>
        <LM label="NP"><w.rf>t#s2w3</w.rf></LM>
        <LM label="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 in Example B.15, “Sample instance after knitting (`example7_knit.xml`)”.

Example B.15: Sample instance after knitting

```
<?xml version="1.0"?>
<annotation xmlns="http://ufal.mff.cuni.cz/pdt/pml/">
  <head>
    <schema>
      <s:pml_schema xmlns:s="http://ufal.mff.cuni.cz/pdt/pml/schema/" version="1.1">
 <s:import schema="example7_schema.xml"/>
 <s:derive type="node.type">
   <s:structure>
     <s:member name="w"><s:list ordered="0" type="w.type"/></s:member>
     <s:delete>w.rf</s:delete>
   </s:structure>
 </s:derive>
      </s:pml_schema>
    </schema>
    <references>
      <reffile name="tokenization" id="t" href="example6.xml"/>
    </references>
  </head>
  <S sentence.rf="s1">
    <LM label="NP"><w id="s1w1">John</w></LM>
    <LM label="VP">
      <w id="s1w2">loves</w>
      <constituents label="NP">
        <w id="s1w3">Mary</w>
      </constituents>
    </LM>
  </S>
  <S sentence.rf="s2">
    <LM label="NP"><w id="s2w1">He</w></LM>
    <LM label="VP">
      <w id="s2w2">told</w>
      <constituents>
        <LM label="NP"><w id="s2w3">her</w></LM>
        <LM label="ADVP">
          <w>
            <LM id="s2w4">this</LM>
            <LM id="s2w5">Friday</LM>
          </w>
        </LM>
      </constituents>
    </LM>
  </S>
</annotation>
```

5. Modular schemas

In the last schema example `example7_schema.xml`, we have repeated the type `w.type` from `example6_schema.xml`. In real-world cases, copy-paste repetitions lead to many maintainability issues and are not a good practice. Starting from version 1.1, PML provides support for modularization of PML schemas via `import` and `derive` instructions. Here is how `example7_schema.xml` schema could be rewritten using these features:

Example B.16: Modular version of the schema example7_schema.xml

```
<?xml version="1.0"?>
<pml_schema version="1.1" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <revision>0.7</revision>
  <description>
    Example of tree annotation over a tokenization layer
  </description>
  <reference name="tokenization" readas="dom"/>

  <import type="w.type" schema="example6_schema.xml" revision="0.2"/>

  <!-- after DOM-knitting, #ID attribute contains a PMLREF -->
  <derive type="w.type">
    <container>
      <attribute name="id" role="#ID" required="1">
 <cdata format="PMLREF"/>
      </attribute>
    </container>
  </derive>

  <root name="annotation" type="annotation.type"/>

  <type name="annotation.type">
    <sequence role="#TREES">
      <element name="S" type="S.type"/>
    </sequence>
  </type>
  <type name="S.type">
    <container role="#NODE">
      <attribute name="sentence.rf">
 <cdata format="PMLREF"/>
      </attribute>
      <list ordered="1" role="#CHILDNODES" type="node.type"/>
    </container>
  </type>
  <type name="node.type">
    <structure role="#NODE">
      <member as_attribute="1" name="label" type="label.type"/>
      <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="label.type">
    <choice>
      <value>S</value>
      <value>VP</value>
      <value>NP</value>
      <value>PP</value>
      <value>ADVP</value>
      <!-- etc. -->
    </choice>
  </type>
</pml_schema>
```

The modularity allows for extending the imported schema. Example B.17, “Extending PML schemas (example9_schema.xml).” presents a PML schema based on example8_schema.xml but largely extended. It also borrows a single type from example1_schema.xml.

Example B.17: Extending PML schemas

```
<?xml version="1.0"?>
<pml_schema version="1.1" xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/">
  <revision>0.1</revision>
  <description>
    Extended example of tree annotation over a tokenization layer
  </description>
  <reference name="tokenization" readas="dom"/>

  <import schema="example8_schema.xml"
          minimal_revision="0.4"
          maximal_revision="1.0"/>
  <import schema="example1_schema.xml" type="meta.type"/>

  <derive type="annotation.type">
    <sequence content_pattern="meta, S+">
      <element name="meta" type="newmeta.type"/>
    </sequence>
  </derive>
  <derive type="S.type">
    <container>
      <attribute name="annotators_comment">
        <cdata format="any"/>
      </attribute>
    </container>
  </derive>
  <derive type="meta.type" name="changes.type">
    <structure>
      <member name="id" role="#ID" as_attribute="1" required="1">
        <cdata format="ID"/>
      </member>
      <member name="desc"><cdata format="any"/></member>
    </structure>
  </derive>
  <derive type="label.type">
    <choice>
      <value>SDECL</value>
      <value>SIMP</value>
      <value>SQUEST</value>
      <delete>S</delete>
    </choice>
  </derive>
  <type name="newmeta.type">
    <structure>
      <member name="lang"><cdata format="any"/></member>
      <member name="changes"><list type="changes.type" ordered="1"/></member>
    </structure>
  </type>
</pml_schema>
```

Modular PML schemas can be converted back to non-modular, simplified, PML schemas for easier processing. The schema in Example B.18, “Extending PML schemas (example10_schema.xml).” is the simplified version of example9_schema.xml, automatically generated by the tool pml_simplify, mentioned in Section 10.2, “Tools”.

Example B.18: Extending PML schemas

```
<?xml version="1.0"?>
<!--
  Created by pml_simplify on Thu Apr  8 22:20:22 2010
  Command-line: pml_simlify examples/example9_schema.xml
-->
<pml_schema xmlns="http://ufal.mff.cuni.cz/pdt/pml/schema/" version="1.1">
  <revision>0.1</revision>
  <description>
    Extended example of tree annotation over a tokenization layer
  </description>
  <reference name="tokenization" readas="dom"/>
  <!--  ============ import schema="example8_schema.xml" ============  -->
  <root name="annotation" type="annotation.type"/>
  <type name="w.type">
    <container>
      <attribute name="id" role="#ID" required="1">
        <cdata format="PMLREF"/>
      </attribute>
      <cdata format="any"/>
    </container>
  </type>
  <type name="ID.type">
    <cdata format="ID"/>
  </type>
  <!--  ============ derived from annotation.type ============  -->
  <type name="annotation.type">
    <sequence role="#TREES" content_pattern="meta, S+">
      <element name="S" type="S.type"/>
      <element name="meta" type="newmeta.type"/>
    </sequence>
  </type>
  <!--  ============ derived from S.type ============  -->
  <type name="S.type">
    <container role="#NODE">
      <attribute name="annotators_comment">
        <cdata format="any"/>
      </attribute>
      <attribute name="sentence.rf">
        <cdata format="PMLREF"/>
      </attribute>
      <list ordered="1" role="#CHILDNODES" type="node.type"/>
    </container>
  </type>
  <type name="node.type">
    <structure role="#NODE">
      <member as_attribute="1" name="label" type="label.type"/>
      <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>
  <!--  ============ derived from label.type ============  -->
  <type name="label.type">
```

```
    <choice>
      <value>VP</value>
      <value>NP</value>
      <value>PP</value>
      <value>ADVP</value>
      <!-- etc. -->
      <value>SDECL</value>
      <value>SIMP</value>
      <value>SQUEST</value>
    </choice>
  </type>
  <!--  =============================================================  -->
  <!--  ============ import schema="example1_schema.xml"  type="meta.type"============  -->
  <type name="meta.type">
    <structure>
      <member name="annotator">
        <cdata format="any"/>
      </member>
      <member name="datetime">
        <cdata format="any"/>
      </member>
    </structure>
  </type>
  <!--  ==============================================================================  -->
  <type name="newmeta.type">
    <structure>
      <member name="lang">
        <cdata format="any"/>
      </member>
      <member name="changes">
        <list type="changes.type" ordered="1"/>
      </member>
    </structure>
  </type>
  <!--  ============ derived from meta.type ============  -->
  <type name="changes.type">
    <structure>
      <member name="annotator">
        <cdata format="any"/>
      </member>
      <member name="datetime">
        <cdata format="any"/>
      </member>
      <member name="id" role="#ID" as_attribute="1" required="1">
        <cdata format="ID"/>
      </member>
      <member name="desc">
        <cdata format="any"/>
      </member>
    </structure>
  </type>
</pml_schema>
```