Customizing Akoma Ntoso: modularization, restrictions, extensions

Fabio Vitali
University of Bologna

Executive summary

In this document we introduce and explain a few mechanisms for correctly creating custom versions of the Akoma Ntoso schema, in order to provide a better fit of the needs and document types of a specific community or context or nationality, without loosing validity, interoperability, and support from standard Akoma Ntoso concepts and tools.

The Akoma Ntoso language is expressed in XML Schema, which is a validation language that does provide a rich and sophisticated set of methods for creating derived schemas from a principal one. These methods are of course always available to the expert user of XML Schema, but dangers exist with careless derivations of Akoma Ntoso using plain XML Schema methods, and on the other hand easy mechanisms for customization were built in in Akoma Ntoso already, that require no special XML Schema expertise or additional schema files.

Thus, in this document we discuss only derivations that generate *correct* custom schemas of Akoma Ntoso (we will try in the section 1 to be more precise about the intuitive idea of correctness in this context) and the discussion will introduce the different approaches in order of complexity and impact on schema and tools, ranging from schema-less restrictions (that are very easy to deal with, in section 2), through schema-less extensions (which are also very easy, but rapidly show the intrinsic limitation of extensions, as explained in section 3) and rapidly ramping up with difficulties and risks introducing the mechanism to actually create derived schemas, in section 4. A different issue is translation, which cannot be accepted as a customization of the language (clearly it creates a different vocabulary) but may be required in specific contexts. This document does not justify translations of Akoma Ntoso, but provides the least onerous path to which such activity can take place, despite acknowledging that it has the highest impact on the schemas and the tools and in fact requires continuous and complex efforts in synchronization and compliancy. It is clearly not something for weak hearts. These topics are discussed in the appendix to this document.

Table of content

1 The customization of XML schemas

An XML Schema is a contract between producers and consumers of a certain class of documents: a set of rules that the producers vow to comply to, and that the consumer expects compliance of. The purpose of such contract is to confirm the consumer that it will not be necessary to design support for documents that deviate from the norms expressed in it.

A custom XML schema designed especially for a specific producer and a specific consumer usually provides exactly the necessary compliancy rules, no less, no more. Similarly, a producer that controls exactly and completely the production process can provide the document exactly in the form required by the consumer and the schema becomes more like a blueprint than a commitment.

With real life parliamentary documents, unfortunately, neither opportunity happens: the producer of XML representation of such documents (e.g., the drafting office of a Parliament) does not have a direct connection to its consumers, which may be separated by space, time and purpose; similarly, the producer is also not usually in a position to command the content author (the legislator) to adhere to the strict set of policies that would make its output regular, consistent and predictable.

This goes to justify why a schema aiming at generality and universality such as Akoma Ntoso may induce some producers or consumers to look for ways to tighten up the validation rules contained in it: maybe the schema is too general, and combination of elements exist that are valid according to the schema but that the consumer is not willing to accept, or maybe the schema is incomplete, and specific elements or combination of elements are not contemplated while the consumer would accept and even require them.

The two situations described above sum up the basic variety of customization options available on an XML Schema:

- a *restriction* is a derived rule that is stricter than the base one, so that all documents that are correct according to the derived rule are also valid according to the base one: the set of valid documents for the derived rule is a mathematical subset of the set of valid documents according to the base rule.
- an *extension* is a derived rule that adds in a controlled way new features to a base rule, so that all documents that are correct according to the derived rule can be validated against the base rule by removing the additional features. The set of valid documents according to the derived rule is a controlled superset of the set of valid documents according to the base rule.

Akoma Ntoso provides simple ways to do both restrictions and extensions in a controlled and easy way without the need to manually create a custom schema, by providing a few tools to this end. More radical restrictions and extensions are possible by creating a custom schema, where both restrictions and extensions can find a place, and for this reason are described together. The final customization option is the most radical: the full translation of the schema, which requires both a custom schema and a transformation document to verify full compliancy to Akoma Ntoso.

1.1 Schema-less restrictions

The easiest way to do restrictions is simply to avoid using the undesired markup features. We call this option *self-constrained markup*, and although it might sound trivial, it is by far the most effective and simple way to do restrictions. In addition to this, we created a new web-based tool, called *Akoma Ntoso subschema generator (anssg)*, which creates restricted schemas in a simple and controlled environment by selecting and composing predefined modules. This is the easiest mechanism available for creating actual schema files that allow a validation engine to verify the documents against derived rules.

Modules come in preset combinations but can be selected and combined freely, always generating a valid subschema of Akoma Ntoso. A simple XML vocabulary over the global XML Schema allows defining additional modules and preset combinations, so a third mechanism for restrictions is to create one's own modules and combinations by acting on the XML vocabulary.

1.2 Schema-less extensions

Extending does not mean loosening constraints: if an item is required or prohibited in a rule, it must remains required or prohibited in the extended rule. In XML schema, extension only means the inclusion of additional elements in the allowed vocabulary, not the modification or overthrowing of rules over the existing items.

In Akoma Ntoso, there are three easy ways to do extensions without affecting the existing schema. The first caters to the most frequent circumstance for extensions: the need to specify additional, task-dependent or site-dependent metadata. For these situations, a specific metadata element is specified in the Akoma Ntoso grammar, `<proprietary>`, within which it is possible to specify just any collection of metadata with no restrictions on vocabulary, values or number, except that the added element must ***not*** belong to the Akoma Ntoso namespace.

The second way to do extensions affects the actual content, and is appropriate for those (hopefully rare) situations in which the descriptive power of the existing Akoma Ntoso elements is not enough. A number of generic elements are available for this situation, with the understanding that they must be used in the appropriate contexts and with the appropriate content and attributes.

Finally, a specific element, `foreign`, was added to allow the inclusion of content expressed in a XML vocabulary different from Akoma Ntoso. This may refer to mathematical formulae expressed in MathML or to drawings expressed in SVG.

1.3 Custom schemas

Once the above-mentioned options for customizing the Akoma Ntoso schema have been explored and found ultimately insufficient, the next step requires creating one's own version of the schema, either by including the general Akoma Ntoso and specify only the differences, or by duplicating the schema and changing it where needed. This requires careful choices, as there are requirements in the existing schema and in the underlying philosophy that should not be ignored or violated.

In particular, Akoma Ntoso heavily relies on a few fundamental principles that need to be respected in any custom schema, including the fundamental separation between content and metadata, and the basic organization of elements in patterns (containers, hierarchical containers, blocks, inlines and metadata elements). Any custom schema needs to respect these basic principles both in practice and in concept, and use separate namespaces for existing structures and new ones.

1.4 Translations

The most radical customization of Akoma Ntoso is the complete translation of its elements. Translations have the highest impact on Akoma Ntoso that any of the previous customizations, since the existing schema, the existing XSLT stylesheets, and all the existing tools need to be properly and completely adapted for the new language.

In fact, since the language changes completely and potentially no current element in the main version of the language exists in the translation. As such, no translation can be considered a version of Akoma Ntoso. Strictly speaking, it is not a customization of the language, but the invention of a completely new language. A possible path is to provide a narrowly defined path for translations of Akoma Ntoso to be created in such as way to ease and foster alignment and translatability of the documents. For this reason, it is necessary that translations of Akoma Ntoso are associated to a specific namespace, and that a translator (in the form of a XSLT stylesheet, for instance) is provided for every translation.

In the following sections we examine each approach to customization in detail and with concrete syntactical examples. The discussion will have some technical aspects throughout the rest of the document, and in the last pages will inevitably become heavily technical.

2 Restricting Akoma Ntoso without a custom schema

In this section we introduce the three mechanisms for creating restrictions in Akoma Ntoso that require no modifications or additions to the current set of schemas.

Again, it is worthy to remind that a restriction in an XML schema is closely related to the concept of subset in set theory. In particular, a schema identifies implicitly a set of valid documents, and therefore a restriction of that schema must identify a subset of the main set, i.e., every valid document according to the restriction ***must be*** a valid document according to the main schema.

2.1 Self-constrained markup

The first and most obvious type of restriction is simply to avoid using undesired elements and attributes. In fact, since restrictions basically means eliminating from actual document all unneeded optional elements and attributes, it is possible to obtain exactly the desired restriction by not using the optional elements that are not needed in the specific context, without affecting the schema at all.

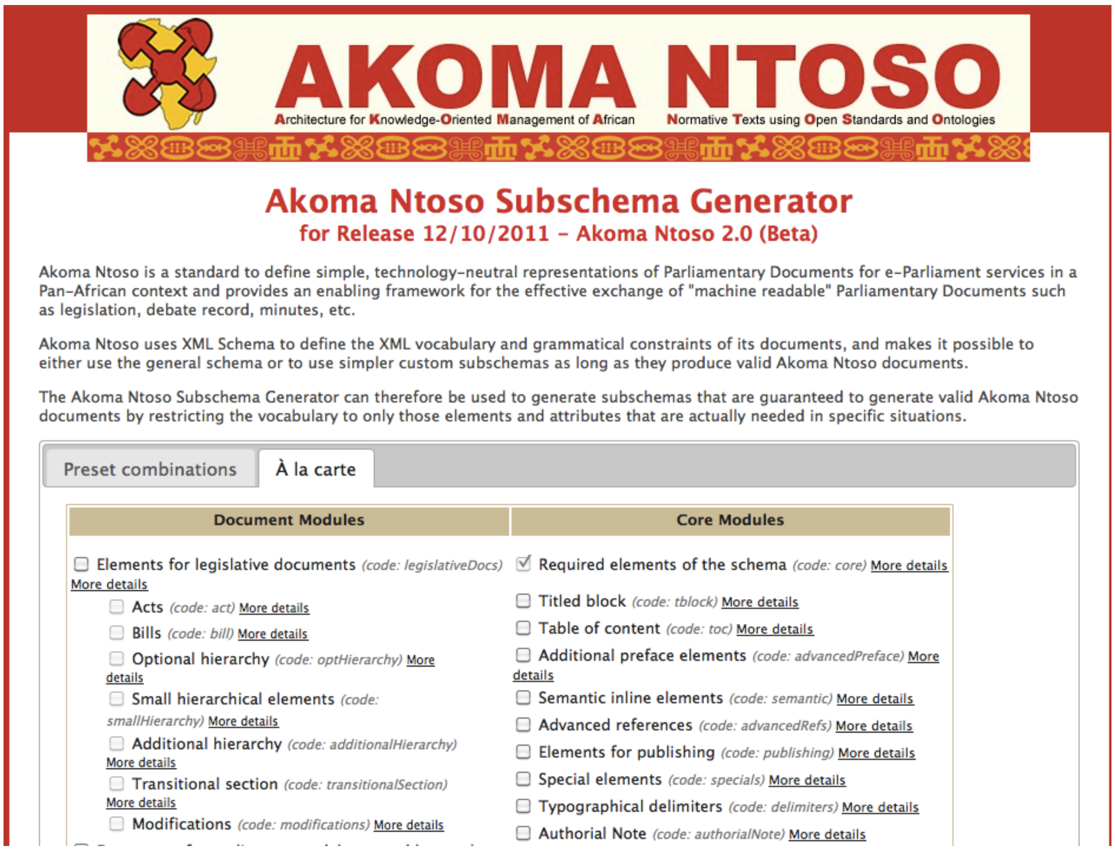
On the other hand, the main reason of using a schema is to have a mechanism that verifies whether the XML documents follow the specified rules, and self-constrained markup has no supporting tool to verify whether the self-constraint is actually working. In this sense, therefore, self-constrained markup makes sense either when the individual author of the XML markup is also the author of the self-constraint rules, or when it is possible to implement the self-constraint rules in a separate tool, e.g. an XML editor.

In this case, the tool will never allow the author of the markup to select and use the undesired elements and attributes, and the restriction becomes possible in a completely automatic way.

2.2 Selecting combinations and individual modules

Starting with version 2.0 of the Akoma Ntoso vocabulary, a new modular architecture of the schema has been introduced, and a tool with it that allows anyone to create custom schemas by selecting individual modules of the vocabulary. The tool is called “Akoma Ntoso Sub Schema Generator” (AKNSSG) and is shown in figure 1 and 2. A test installation is available at the URI http://akn.web.cs.unibo.it/aknssg/aknssg.html.

Currently there are 26 modules, as shown in fig. 1. The core set is required, and the user can choose optional modules. They are organized firstly in document types (legislation, reports, amendments, judgments, collections) and then around optional features (specific elements such as titled blocks, tables of content, etc.).



AKOMA NTOSO
Architecture for Knowledge-Oriented Management of African Normative Texts using Open Standards and Ontologies

Akoma Ntoso Subschema Generator
for Release 12/10/2011 – Akoma Ntoso 2.0 (Beta)

Akoma Ntoso is a standard to define simple, technology-neutral representations of Parliamentary Documents for e-Parliament services in a Pan-African context and provides an enabling framework for the effective exchange of "machine readable" Parliamentary Documents such as legislation, debate record, minutes, etc.

Akoma Ntoso uses XML Schema to define the XML vocabulary and grammatical constraints of its documents, and makes it possible to either use the general schema or to use simpler custom subschemas as long as they produce valid Akoma Ntoso documents.

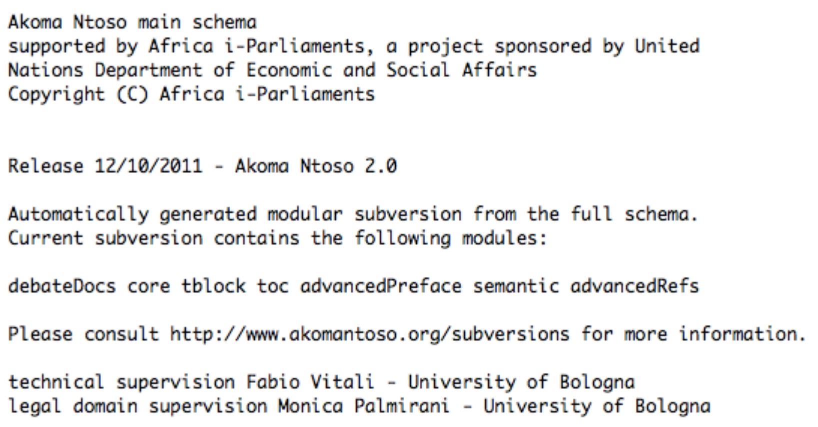
The Akoma Ntoso Subschema Generator can therefore be used to generate subschemas that are guaranteed to generate valid Akoma Ntoso documents by restricting the vocabulary to only those elements and attributes that are actually needed in specific situations.

Preset combinations | À la carte

Document Modules	Core Modules
☐ Elements for legislative documents *(code: legislativeDocs)* More details	✓ Required elements of the schema *(code: core)* More details
☐ Acts *(code: act)* More details	☐ Titled block *(code: tblock)* More details
☐ Bills *(code: bill)* More details	☐ Table of content *(code: toc)* More details
☐ Optional hierarchy *(code: optHierarchy)* More details	☐ Additional preface elements *(code: advancedPreface)* More details
☐ Small hierarchical elements *(code: smallHierarchy)* More details	☐ Semantic inline elements *(code: semantic)* More details
☐ Additional hierarchy *(code: additionalHierarchy)* More details	☐ Advanced references *(code: advancedRefs)* More details
☐ Transitional section *(code: transitionalSection)* More details	☐ Elements for publishing *(code: publishing)* More details
☐ Modifications *(code: modifications)* More details	☐ Special elements *(code: specials)* More details
	☐ Typographical delimiters *(code: delimiters)* More details
	☐ Authorial Note *(code: authorialNote)* More details

fig. 1: the modules’ interface of the Akoma Ntoso SubSchema Generator

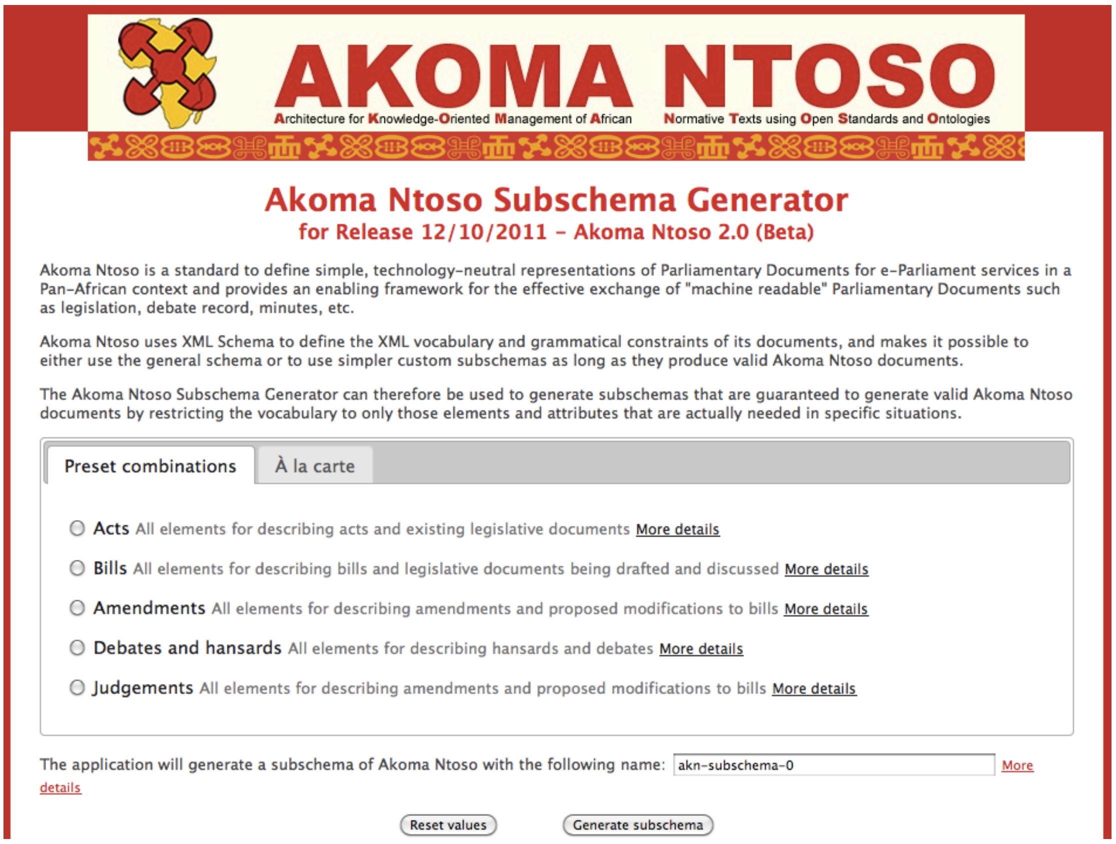
Each module is associated to a name, a code, and to a list of features (i.e., of elements and attributes) it contains. Each selection of modules creates a subschema of the general schema, and lists the modules chosen for its generation, as in fig. 2, after the sentence “Current subversion contains the following modules:”.



```
Akoma Ntoso main schema
supported by Africa i-Parliaments, a project sponsored by United
Nations Department of Economic and Social Affairs
Copyright (C) Africa i-Parliaments


Release 12/10/2011 - Akoma Ntoso 2.0

Automatically generated modular subversion from the full schema.
Current subversion contains the following modules:

debateDocs core tblock toc advancedPreface semantic advancedRefs

Please consult http://www.akomantoso.org/subversions for more information.

technical supervision Fabio Vitali - University of Bologna
legal domain supervision Monica Palmirani - University of Bologna
```

fig. 2 – The header of a subschema with the list of selected modules

To further simplify the creation of subschema, predefined combinations have been created for the most common situations. They are available in a different page of the same application, shown in fig. 3.



AKOMA NTOSO
Architecture for Knowledge-Oriented Management of African Normative Texts using Open Standards and Ontologies

Akoma Ntoso Subschema Generator
for Release 12/10/2011 – Akoma Ntoso 2.0 (Beta)

Akoma Ntoso is a standard to define simple, technology-neutral representations of Parliamentary Documents for e-Parliament services in a Pan-African context and provides an enabling framework for the effective exchange of "machine readable" Parliamentary Documents such as legislation, debate record, minutes, etc.

Akoma Ntoso uses XML Schema to define the XML vocabulary and grammatical constraints of its documents, and makes it possible to either use the general schema or to use simpler custom subschemas as long as they produce valid Akoma Ntoso documents.

The Akoma Ntoso Subschema Generator can therefore be used to generate subschemas that are guaranteed to generate valid Akoma Ntoso documents by restricting the vocabulary to only those elements and attributes that are actually needed in specific situations.

Preset combinations | À la carte

- ○ Acts All elements for describing acts and existing legislative documents More details
- ○ Bills All elements for describing bills and legislative documents being drafted and discussed More details
- ○ Amendments All elements for describing amendments and proposed modifications to bills More details
- ○ Debates and hansards All elements for describing hansards and debates More details
- ○ Judgements All elements for describing amendments and proposed modifications to bills More details

The application will generate a subschema of Akoma Ntoso with the following name: akn-subschema-0 More details

fig. 3: The combinations' interface of the Akoma Ntoso SubSchema Generator

By clicking on "more details" it is possible to read the list of selected modules for each preset combination.

2.3 Creating new combinations and new modules

Of course the selection of the name and content of each preset combination is a design choice of much lesser importance than the details of the schema itself. Thus for instance the current choice of preset combinations or the decision of whether, say, the “amendments” combination should contain the “special elements” module, is a matter of taste and sensibility that may very well find itself ungrounded or inappropriate in other situations.

Even the current list of modules is nothing more that a reasonable choice of elements and attributes for specific purposes. Their number and content is not only easier to dispute and modify than the overall schema (specific requests in that direction sent to akomantoso-xml@googlegroups.com would be implemented rapidly and with little discussion), but it also requires little specific competency of XML schema or DTD++ and can be done with little effort. This requires just some competencies in XML and the overall structure of the Akoma Ntoso schema.

The modular architecture of Akoma Ntoso relies on an XML that uses no namespace and contains exactly 5 XML elements:

- `modular`: the root element.
- `combos`: the list of available preset combinations.
- `combo`: the definition of a preset combination. It has attributes `id`, `name` (used in the web interface for the identification of the combination), `desc` (a string shown when the “show more” link is selected) and `content` (a list of the ids of the include modules contained in the preset combination). For instance, the preset combination for the Act is as follows:

```
<combo id="acts" name="Acts" desc="All elements for describing acts and existing legislative documents" content="core legislativeDocs act modifications tblock semantic advancedRefs authorialNote specials delimiters table"/>
```

- `report`: Report is the place in the document where the actual list of modules is shown. It is currently present in one place only, in the initial large comment describing the release. The following report element:

  ```
  <report version="Release 12/10/2011 - Akoma Ntoso 2.0"/>
  ```

  is shown in the final schema as follows for the complete version:

  ```
  Release 12/10/2011 - Akoma Ntoso 2.0
  Complete version.
  ```

while for a subschema (in this case the *minutes* subschema) automatically renders as:

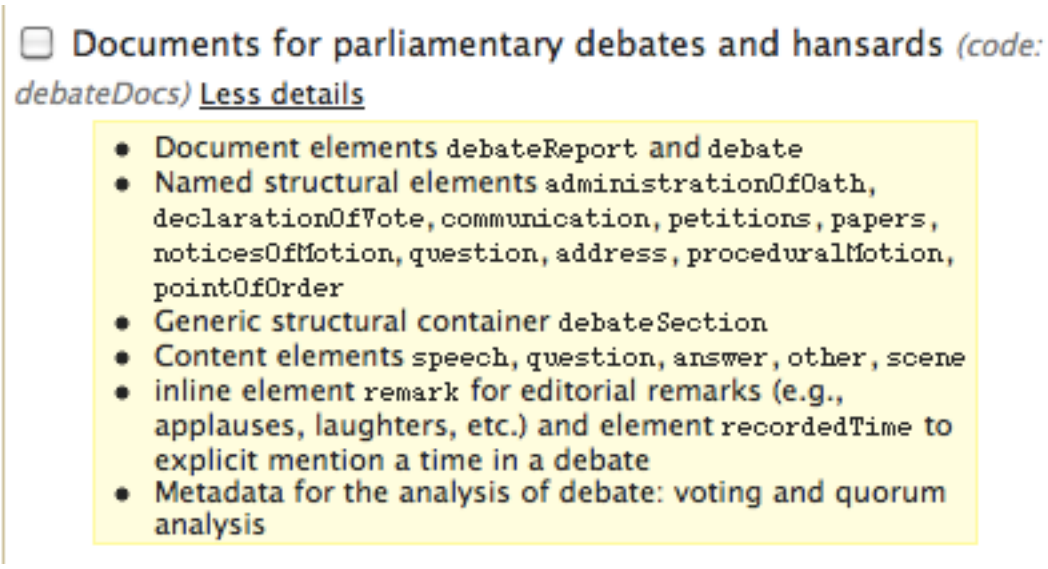
```
Release 12/10/2011 - Akoma Ntoso 2.0
Automatically generated modular subversion from the full schema.
Current subversion contains the following modules:
debateDocs    core    tblock    toc    advancedPreface    semantic
advancedRefs
```

- `include`: each individual preset combination. It has attributes `label`, `desc` (for description), `if` and `v` (for value). Consider for instance the two following examples
  <include if="debateDocs" label="Documents for parliamentary debates and hansards" desc="Document elements $debateReport and $debate">…</include>

  <include if="debateDocs" v="…" />

  The `if` attribute determines the modules to which each document fragment belongs to. Thus in this example both `include` structures belong to the same module, *debateDocs*. There is no limit to the number and position of `include` elements having the same value for their `if` attribute: all of them will belong to the same module.

  The `label` attribute contains the short name describing the modules, and is used in the interface of the Akoma Ntoso SubSchema Generator for the identification of the module. Similarly, the content of each `desc` attribute is shown as one bullet in the full list shown when the corresponding "show more" link is clicked, as follows:



☐ Documents for parliamentary debates and hansards *(code: debateDocs)* Less details

- Document elements `debateReport` and `debate`
- Named structural elements `administrationOfOath`, `declarationOfVote`, `communication`, `petitions`, `papers`, `noticesOfMotion`, `question`, `address`, `proceduralMotion`, `pointOfOrder`
- Generic structural container `debateSection`
- Content elements `speech`, `question`, `answer`, `other`, `scene`
- inline element `remark` for editorial remarks (e.g., applauses, laughters, etc.) and element `recordedTime` to explicit mention a time in a debate
- Metadata for the analysis of debate: voting and quorum analysis

Fig. 4: a detail of the on screen representation a module in the subschema generator

As can be seen, the display automatically shown with a different font all word beginning with the “$” character, which is used to point out the element names described in the module.

There are two types of `include` elements: those including larger portions of text of the full schema (such as the first example), whose content is directly shown within the start and end tags of the `include` element, and those that contain only small fragments of text (as in the second example). These are only composed of an empty `include` element where the text to be included is specified by the `v` (for value) attribute. They are used to specify, within more complex structure where the fragment is present, just the few characters that need to be included or excluded if the corresponding module is selected or not.

Creating new combinations is therefore very easy: one needs only to add another `combo` element within the `combos` structure at the beginning of the schema, and list in the `content` attribute all the ids of the modules that belong to it. After placing the full modular schema in the appropriate directory of the Akoma Ntoso SubSchema Generator, the new combination is shown in the preset tab and can be chosen to create the corresponding subschema.

To create a new module, one needs to create as many `include` elements as there are fragments of text, throughout the full schema, that needs to be included or excluded from the subschema if the module is selected or not. This refers to both whole sections of the schemas, e.g., where the elements and their attributes are defined, using plain include tags to contain them, as well as smaller fragments of the schema within the types and element groups that include these elements and attributes. In this case, the empty version of the include element should be used, specifying in the `v` attribute the few characters that should be included or excluded. The existing schema has plenty of examples where both approaches are taken, and should be used as example.

3 Extending Akoma Ntoso without a custom schema

In this section we introduce the three mechanisms for creating extensions to the Akoma Ntoso vocabulary that require no modifications or additions to the current set of schemas.

In general, an XML schema provides both a vocabulary (of elements and attributes) and a set of rules that these elements should undergo. A lawful extension to an XML schema means providing more vocabulary than in the original one, but it **does not** mean changing the set of rules, i.e., allowing structure and uses of the old vocabulary that in the original schema were prohibited. Thus the features of any extension of the Akoma Ntoso will never impact existing elements and attributes, which will never be allowed to be used in a different way than the one sanctioned in the original schema, but will only introduce new elements and attributes.

In this section we only discuss extensions that do not require modifications to the schema, but are available now with the current schemas (or, to be more precise, that are available with the inclusion of specific modules that belong to the full schema, but may not be selected in specific subschemas). By definition, all schema-less extensions of an XML schema cannot introduce new rules (which would require the schema to be amended), and thus the extensions listed in the following cannot be constrained, as long as they are used as described here.

Akoma Ntoso strongly differentiates metadata and content, both conceptually and physically (all metadata is contained within a section at the beginning of the document, separate from the markup of the document's content). Thus there exist two very different ways to provide extensions to the schema for metadata and content.

3.1 Adding new metadata

Akoma Ntoso does NOT allow the existing, pre-defined metadata elements to be extended or used in a different way, but provides a subsection of the metadata section of the document where any new metadata element can be added without constraints.

Consider the following Akoma Ntoso document:

```
<?xml version="1.0" encoding="UTF-8"?>
<akomantoso xmlns="http://www.akomantoso.org/2.0">
   <act contains="OriginalVersion">
      <meta>
         <identification source="#au1"> ... </identification>
         <publication … />
         <lifecycle source="#au1"> ... </lifecycle>
         <analysis source="#au1"> ... </analysis>
         <references source="#au1"> ... </references>
         <proprietary source="#au1"
```

```
                    xmlns:cirsfid="http://www.cirsfid.unibo.it/proprietary">
                <cirsfid:MissingInfo>
                    <cirsfid:mActDate>1992-12-28</cirsfid:mActDate>
                </cirsfid:MissingInfo>
            </proprietary>
        </meta>
        <preface>
            <p><ActTitle id="ActTitle">Copyright (Amendment) Decree</ActTitle>
                <eol />No <ActNumber>98</ActNumber> of 1992<eol /></p>
        </preface>
        <preamble id="preamble"> ... </preamble>
        <clauses> ... </clauses>
    </act>
</akomantoso>
```

Here the editor of the document, the organization CIRSFID (University of Bologna), wants to record information that is not expressible through the vocabulary of Akoma Ntoso, i.e., the indication of the document date when it is not explicitly present in the content of the document. Since this information is not part of the content, it must become an editorial annotation, i.e., it is clearly part of the metadata. Correctly, this information is placed within the `proprietary` block of the `metadata` section of the document.

Akoma Ntoso makes no restriction as to the vocabulary or containment constraints of the content of the `proprietary` block. The only requirement (correctly complied to by the fragment shown) is that all elements belong to a namespace different than that of Akoma Ntoso. This guarantees the immediate identification of the new elements.

Proprietary elements can still use existing core Akoma Ntoso attributes and probably should, whenever they need to be used for the reason they were defined in the first place. In particular, the following attributes should be used whenever possible even in proprietary elements:

- `id`: an identification word unique within the whole document
- `refersTo`: the id of an ontological concept, person, organization, location etc. defined in the `reference` section of the metadata.
- `href`: the address of another document, or the id of a section of the document referenced by the individual metadata element.
- `source`: the id of a person or organization (placed in the `reference` section) providing the metadata element.

3.2 Generic elements

Akoma Ntoso fully embraces the philosophy of descriptiveness intrinsic in the XML language. This means that the vocabulary of Akoma Ntoso is very rich (275 elements and 92 attributes in the October 2011 version) and aims at supporting precisely the descriptive needs of document authors.

Nonetheless, it is clear that there are cases and situations that could not be foreseen when designing the vocabulary, and that would require specific descriptive elements that do not exist in the current version of the Akoma Ntoso language. For these situations there exists a workaround based on the notion of generic element.

A generic element is an element with a generic name and an attribute where its would-be name can be specified, that requires a very precise content model. Once it is determined that an element with the required name does not exist in the vocabulary, a generic element can be used instead, taking good care that the chosen generic element has the right content model. By specifying different would-be names within the `name` attribute, one can therefore introduce as many new elements as needed, generating a *de facto* extension of the language without the need to introduce new elements.

In fact, Akoma Ntoso is very generous with element names, but very parsimonious in content models (six patterns account for the content models of most elements), so that the number of different behaviors when dealing with elements is fairly limited. Given this, it is a requirement for a correct extension of Akoma Ntoso that the correct content model is identified among the few available, and the corresponding generic element is chosen.

As explained in section 4.1 of the Akoma Ntoso release notes, there are six patterns in content models:

- The *markers* are content-less elements placed in the content of the document that are meaningful for their position, their names and their attributes.
- The *inline* elements are element placed within a mixed model element (e.g. a block) to identify a small fragment of text as relevant for some reason. There are both semantically relevant inlines and presentation oriented inlines.
- The *block* elements are containers of text, inlines, and markers, and their content is organized vertically on the display (i.e., has paragraph breaks).
- The *popup* elements are elements placed within a mixed model element that identify a completely separate context that, for any reason, appears within the flow of the text, but does not belong to it or does not follow its rules. Popup elements are containers appearing in the middle of sentences but containing full structures (with no direct containment of text or inline elements).

- The *hierarchical container* elements are a set of sections nested to an arbitrary depth, all provided with title and numbering. Each level of the nesting can contain either more nested sections or a container.
- The *container* elements are sequences of other elements, some of which can be optional. Containers are all different from each other (since the actual list of contained elements vary), and so there is no single container content model, but rather a number of content models that share the same conceptual category.

For each of these patterns there exists a corresponding generic element called exactly as the pattern. The `name` attribute *must* be used to provide a description of the element, with the explicit requirement that the name attribute should be the name of the element if it were to be accepted in the language as an extension.

For instance, if there were the need for an element wrapping a small fragment of text within a sentence to identify, say, the first name of an individual, then the correct content model would be inline, and therefore the `inline` generic element should be used. The following could very well be a correct solution:

```
<inline name="firstName">Fabio</inline>
```

Besides the generic elements corresponding to the six basic content model patterns, there are a few other generic elements that can be used for extensions of other structured elements according to need:

- `doc` and `documentCollection`: two generic root elements for all types of documents (respectively, collections of documents) that are not explicitly handled by more specific document elements, such as `act`, `debateRecord`, `judgement`, etc.
- `debateSection`: Akoma Ntoso has a collection of some 15 different section names that can be used in the report of a debate. This generic element can be used for section names that are not included in these 15.
- `entity` and `TLCReference`: all Top Level Classes of the Akoma Ntoso ontology have their own specific inline elements (such as `person`, `organization`, etc.). For those needs to mark elements with a specific connection to the ontology, but whose class is not among the TLC ones, `entity` is the right generic element to place in the content of the document, and `TCLReference` for the specification in the reference section.

3.3 The element `foreign`

Akoma Ntoso does not provide markup for situations that are very specific and for which better-suited vocabularies exist already. For instance, mathematical formulas or drawings have well-known standard XML vocabularies that should be used rather than inventing a new one. For these situations, the foreign element can be used to specify fragments of content that correspond to structures and data that are not currently managed by Akoma Ntoso.

For instance, the following is a valid Akoma Ntoso fragment to show the equation ax^2+bx+c:

```
<foreign>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
        <mrow>
            <mi>a</mi>
            <mo>&#x2062;</mo>
            <msup>
                <mi>x</mi>
                <mn>2</mn>
            </msup>
            <mo>+</mo>
            <mi>b</mi>
            <mo>&#x2062;</mo>
            <mi>x</mi>
            <mo>+</mo>
            <mi>c</mi>
        </mrow>
    </math>
</foreign>
```

The foreign element is a block-level element, which means that it is presumed that its content appears in a vertically isolated block. Furthermore, only fully qualified XML fragments can appear there, and they must belong to a different namespace than Akoma Ntoso's. Finally, it should be reminded that foreign should only be used when Akoma Ntoso does not provide support for the notation, and not to simply include XML fragments from other vocabularies. Thus, it is probably not appropriate to place HTML fragments here, as there is no feature of HTML that cannot be expressed in Akoma Ntoso.

4 Creating custom schemas

In this section we examine those customizations to the general Akoma Ntoso schema that imply performing actual modifications to the actual schema. Once we begin editing the actual Akoma Ntoso schema, it becomes possible to add here, remove there, and still come out with a valid XML Schema. Of course, even if the majority of the content in the edited schema derives from the original schema, this is not enough to guarantee that the result is a valid customization of Akoma Ntoso, which has strong requirements for compliance. For this reason, these customizations are best left to an expert in XML Schema and Akoma Ntoso, as it is very easy to generate either an incorrect XML Schema or a correct XML Schema for a language that is not compliant with Akoma Ntoso.

In general and in absolute, the ***fundamental rule for the customization of Akoma Ntoso*** is that *any document that is correct with regard to the custom rules must be also correct (in validity and spirit) with regard to the full Akoma Ntoso schema*.

Concretely, the customization possibilities are limited to what was described in the two previous sections: restrictions of content models and allowed values, and extensions using `proprietary`, `foreign` and generic elements. So, where is the difference now?

Except of course for subschemas generated through the subschema generator, the basic problem that the schema-less customizations present, is that there is no way to verify that the documents comply with the custom schema. The schema is in fact exactly the tool that verifies whether the document complies with the rules of the language, and schema-less customizations have no way to perform checks on the customized parts.

Therefore, if the customization needs exceed those offered by the subschema generator, and it is important to validate the documents with respect to the custom rules, there is no other way but to generate a custom schema. There are three ways to do so:

- *Directly editing the schema*: working directly on a copy of the official schema. This is by far the easiest approach, and, as long as customizations are done correctly, it allows complete control not only on the custom rules, but also on how they interact with the standard Akoma Ntoso rules. The drawback here is that it is very difficult to maintain alignment with the new releases of the base Akoma Ntoso language.
- *Redefining the schema*: creating a separate XML schema that explicitly redefines parts of the official schema, using the `<xsd:redefine>` structure of XML Schema: alignment issues become easier to deal with, since no modification is required in the official schema. On the other hand, the

focus of action is now separated, and the interactions between unchanged Akoma Ntoso structures in the base schema and redefined ones in the custom schema may rapidly become overly complex. Furthermore, restrictions in the redefinition of XML schemas may limit of modifications that this approach allows.

- *Multiple-schema validation*: creating a fully separate schema and validating with both the original schema and the new one, enacting validation as a two-step process: this approach is very safe, very easy to maintain aligned, and allows the use of different languages than XML Schema to provide additional rules, including, for instance, Schematron. On the other hand, it requires building a software infrastructure of validation that is inevitably more complex than with the other approaches.

An advantage of the direct editing of a copy of the Akoma Ntoso schema is its simplicity, but this comes with a relevant disadvantage: it becomes pretty much impossible to verify whether the customized schema is a *correct* schema for Akoma Ntoso document, i.e., whether it complies to the fundamental rule expressed above. On the other hand, creating a redefinition of the schema or an additional schema provide also safe customizations, as they basically prevent violations to the fundamental rule. For this reason, in the following all examples will deal with either a redefined schema or an embedded Schematron schema.

In all cases, although opinions may differ on the most elegant way to proceed, the kinds of modifications that are possible are pretty much the same: you can derive types (most often to restrict rather than extend, as there are many more constraints in Akoma Ntoso with regard to extensions), you can define new attributes (in a different namespace) for any existing elements, or you can define new elements (in a different namespace) within the existing elements that allow them, such as `proprietary` or `foreign`.

4.1 Custom types

The first and most evident customization of the schema is the redefinition of existing types. Regardless of whether one creates a new type, restricts an existing type or edits the main Akoma Ntoso definition of the existing type, the basic requirement is that the fundamental rule for the customization of Akoma Ntos holds: the resulting type must make sure that valid documents for the custom schema are valid documents for the main schema.

In the following, for instance, we redefine (using approach 2, redefining the schema) the base hierarchy types, requiring that elements num and heading become required elements (they are both optional in the full Akoma Ntoso schema):

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" xmlns:an="http://www.akomantoso.org/2.0"
xmlns="http://www.akomantoso.org/2.0"
targetNamespace="http://www.akomantoso.org/2.0">
    <xsd:redefine schemaLocation="./akomantoso20.xsd">
        <xsd:complexType name="basehierarchy">
            <xsd:complexContent>
                <xsd:restriction base="an:basehierarchy">
               <xsd:sequence>
                      <xsd:element ref="num" minOccurs="1" maxOccurs="1"/>
                      <xsd:element ref="heading" minOccurs="1" maxOccurs="1"/>
                      <xsd:element ref="subheading" minOccurs="0" maxOccurs="1"/>
               </xsd:sequence>
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:redefine>
</xsd:schema>
```

In this example, a custom version of the `basehierarchy` type is defined as a restriction of the type found in the full schema. This redefinition is placed within the `xsd:redefine` element, and it has the same name as the base type, so that this definition replaces the original one in all elements using it. By specifying that elements `num` and `heading` become required in all hierarchical elements (`minOccurs='1'` instead of `minOccurs='0'` of the original definition) we have created a concrete, widespread customization of the original schema, and also one that was not possible with the subschema generator.

Please also note that a requirement for redefinitions is that the redefined schema has the same target namespace as the original one, yet that it is possible to distinguish between structures defined in the base schema and structures defined in the custom one. This is possible by associating two different prefixes to the same namespace, i.e. no prefix (xmlns="http://www.akomantoso.org/2.0") and prefix an: (xmlns:an="http://www.akomantoso.org/2.0").

4.2 Custom attributes

While being very rigid about new elements, the Akoma Ntoso schema is much more flexible about custom attributes to existing elements. The rule is that it is possible to create new attributes and assign them to any of the existing elements, as long as these attributes are assigned to a different namespace than Akoma Ntoso.

XML Schema requires that each XSD file is associated to only one namespace (its target namespace), so that the definition of the new attributes ***must be*** in a different namespace than the schema. This means that we need two separate schemas if we edit the Akoma Ntoso schema directly, and three files if we redefine the schema: the original Akoma Ntoso schema, the schema containing the new attributes, and the *pivot schema* that redefines the Akoma Ntoso structures.

Namely, we first need to define the new attributes in a separate schema files associated to a different target namespace, as in the following example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.site.com/" targetNamespace="http://www.site.com/">
            <xsd:attribute name="myAttribute" type="xsd:string"/>
</xsd:schema>
```

Then we need to import the external schema file and use its defined structures. The following is the solution based on redefining the schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" xmlns:an="http://www.akomantoso.org/2.0"
xmlns:x="http://www.site.com/10" xmlns="http://www.akomantoso.org/2.0"
targetNamespace="http://www.akomantoso.org/2.0">
   <xsd:import namespace="http://www.site.com/10" schemaLocation="./site.xsd"/>
    <xsd:redefine schemaLocation="./akomantoso20.xsd">
            <xsd:attributeGroup name="corereq">
                <xsd:attributeGroup ref="an:corereq"/>
                <xsd:attribute ref="x:myAttribute"/>
             </xsd:attributeGroup>
    </xsd:redefine>
</xsd:schema>
```

The file containing the new attributes is first imported with the `<xsd:import>` command, then the redefinition of the Akoma Ntoso schema takes place, where an attribute group (in our case, `corereq`) is redefined by adding to the previously specified definition (`an:corereq`) the new attribute `x:myAttribute`.

Although XML Schema would allow custom attributes to be specified in the Akoma Ntoso namespace, this would go against the rule of the Akoma Ntoso language and should not be performed.

4.3 Custom elements

A shown in the previous section, defining new attributes is easy, and there is no requirement for sequence and organization of the text content of the document. On the other hand, new elements are a completely different problem, because Akoma Ntoso expects a specific sequence of containment when dealing with the actual text content of a document. Custom elements, therefore are only indirectly introduced as generic elements or within special contexts, i.e. those whose type is defined as anyOtherType, a list which includes only one content-oriented element (i.e., foreign) and several metadata element, including `proprietary`, `presentation`, `preservation`, and `otherAnalysis`.

With regard to generic element, a likely customization need could be the constraint of the allowed names, e.g. to verify that only some values for the name attribute are used. As Akoma Ntoso stands now, literally any string can be used as the name for a generic element. If it is important that only a specific new element is used, then the `name` attribute of the corresponding generic element must be limited to its name only.

For instance, the following is a Schematron fragment verifying that only `subsubsection` is used as a value for the `name` attribute of the `hcontainer` generic element.

```
<sch:ns prefix="an" uri="http://www.akomantoso.org/2.0"/>
<sch:pattern id="Subsubsection">
    <sch:rule context="an:hcontainer">
        <sch:report test="@name!='subsubsection'">
            The only generic hierarchical container allowed is 'subsubsection'
        </sch:report>
    </sch:rule>
</sch:pattern>
```

As shown, the Akoma Ntoso namespace is first defined, and then a pattern (i.e., a named set of associated rules activated together for a specific validation need) is created with just one rule, evaluated when encountering and instance of the element `hcontainer`. This checks and reports an error message every time its `name` attribute has a value different than 'subsubsection'.

Similarly, the type `anyOtherType` allows any structure and any element, as long as it uses a different namespace than Akoma Ntoso's. If it is important that only some vocabularies are used, and not just anything (for instance, if we want to restrict the use of `foreign` to only mathematical formulae expressed in MathML), we need to specify a restriction of the extension. The simplest solution is to provide a Schematron rule, since restricting the `anyOtherType` in a redefined schema will affect the allowed values in all elements of that type and not only `foreign`. The following is a suitable Schematron fragment:

```
<sch:pattern id="MathML">
    <sch:rule context="an:foreign">
        <sch:assert
         test="*[namespace-uri()='http://www.w3.org/1998/Math/MathML']">
           The only allowed content for foreign is MathML
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

In this fragment, we simply require every element within the `foreign` element (the context of the rule) to have `http://www.w3.org/1998/Math/MathML` as its namespace.

5 Conclusions

In this document we presented several types of customization possibilities for the Akoma Ntoso schema. Throughout the document, we tried to be very clear of what are the customization options that still leave the affected documents within the Akoma Ntoso document space, as declared by the fundamental rule for customizing Akoma Ntoso: that *any document that is correct with regard to the custom rules must be also correct (in validity and spirit) with regard to the full Akoma Ntoso schema* (page 18). While XML Schema and possibly even Schematron can do much to verify that a customization has been done properly, there are controls that simply cannot be done automatically, and require a careful human eye to validate and approve.

In the appendices, we also point out the basic rule for translations of Akoma Ntoso: *any XML document valid for the translated version of Akoma Ntoso must be* *translated back into a valid Akoma Ntoso document by changing only element and attribute names* (page 23). This, too, is a verification that needs to be performed ultimately by a human expert, but the tools introduced in section 5.4 should considerably reduce the chance of errors and simplify the task.

Appendix A: Translations of Akoma Ntoso

The vocabulary of Akoma Ntoso is mostly in English (except for the root element `akomaNtoso`, which is in Akan, a language from Ghana). Although neither the technical team creating the XML schema nor the initial addressees are English native-speakers, this decision was taken to improve its recognizability and diffusion. Situations may exist, on the other hand, that require a localized version of the Akoma Ntoso vocabulary in a different language. This implies creating a full set of terms for element and attribute names in a different language than English.

Of course, an XML document that uses element names and attribute names that do not belong to the Akoma Ntoso schema ***cannot*** be considered an Akoma Ntoso document: by translating the schema we are getting out of the scope of the Akoma Ntoso language. Yet, of course, there are some translations that are better than others for an Akoma Ntoso environment.

The basic rule for translating Akoma Ntoso is that *any XML document valid for the translated version of Akoma Ntoso must be translated back into a valid Akoma Ntoso document by changing only element and attribute names*.

We need therefore to introduce a class of schemas (and of documents that are validated by them) that, while not being Akoma Ntoso, are close enough to it to be taken into consideration.

The purpose of this section, therefore, is to list first the requirements for a correct translation, and then to describe a simple tool (an XML file and an XSLT stylesheet) that can be used to generate correct translations.

Namespace declaration

An official translation of Akoma Ntoso must target a specific namespace that cannot be the usual Akoma Ntoso namespace, but must still be controlled and not arbitrary. For this reason, official translations of Akoma Ntoso must be associated to namespaces of the form:

```
http://www.akomantoso.org/20/[language]
```

where language is a three-letter code denoting the language according to ISO 639-2 alpha-3 (e.g., English is `eng`, Italian is `ita`, German is `ger`, etc.).

By construction, the full Akoma Ntoso is the English version of Akoma Ntoso, and thus `http://www.akomantoso.org/20/eng` is defined as equivalent to `http://www.akomantoso.org/20`.

Translatable terms

For a schema to be a correct translation of Akoma Ntoso, there must be a conversion mechanism from the translated back into the main Akoma Ntoso vocabulary. This implies that there is a one-to-one mapping between terms of the vocabulary of Akoma Ntoso and terms of the translated vocabulary. These terms include element names, attribute names and enumerations of allowed values in attributes.

Not all terms in the Akoma Ntoso vocabulary can be translated. In particular, the name of the root element `akomaNtoso` must be maintained in all translations, and HTML elements and attributes must be maintained in all translations. These include, among others, elements such as `p`, `b`, `i`, `table`, or `img`, as well as attributes such as `id`, `href`, `class` and `title`.

Translated terms should be correct and currently used terms belonging either to the general language or to the technical and/or juridical vocabulary of the local language. Terms that are composed of exactly one word must be in lowercase, and compositions of different words must be in camelCase, i.e., the term is the collapse of (i.e., the removal of the spaces separating) the words, where the first word is lowercase and all subsequent words have the initial in uppercase.

Finally, no translations should be performed for the two fundamental acronyms that appear in Akoma Ntoso terms: “FRBR” (which stands for “Functional Requirement of Bibliographic Record”), that is the conceptual model on which the document organization of Akoma Ntoso is based; and “TLC” (which stands for “Top Level Class”), that is the ontological framework to connect Akoma Ntoso documents and Semantic Web ontologies. The translation of terms that contain either of the two acronyms must contain them as well.

Other translatable terms

The main Akoma Ntoso schema contains other terms, basically names for complex types, simple types, element groups and attribute groups. There are no requirements on the translation of these terms, but since they do not appear in the final XML document. It is therefore a local decision whether these terms are actually translated or not.

Tools for Akoma Ntoso translations

The main problem with managing Akoma Ntoso translations is the alignment between the original and the translated schema whenever a new release of the language is published. Generating the translation by editing directly the schema is an easy way to do the translation, but requires that it be done again, and exactly, every time a new release of the language becomes available.

We describe here a mapper that simplifies the work for the translators, reduces the possibilities of error or inconsistencies in the translation, and reduces the workload of the synchronization of the translation. The mapper is based on a two-entry vocabulary that lists all translatable terms and their translations. This is an XML file as the following:

```
<akomaNtosoMap xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <source xml:lang="eng" namespace="http://www.akomantoso.org/20" />
    <dest xml:lang="ita" namespace="http://www.akomantoso.org/20/ita" />
    <element source="FRBRExpression" dest="espressioneFRBR" />
    <element source="FRBRItem" dest="elementoFRBR" />
    <element source="FRBRManifestation" dest="manifestazioneFRBR" />
    <element source="FRBRWork" dest="operaFRBR" />
    <element source="FRBRalias" dest="aliasFRBR" />
    <element source="FRBRauthor" dest="autoreFRBR" />
    …
    <attribute source="actor" dest="attore" />
    <attribute source="alternativeTo" dest="alternativoA" />
    <attribute source="as" dest="come" />
    <attribute source="breakat" dest="aCapoPresso" />
    …
    <enumeration source="after" dest="dopo" />
    <enumeration source="agreeing" dest="inAccordo" />
    <enumeration source="amendment" dest="emendamento" />
    …
</akomaNtosoMap>
```

Two applications (really XSLT stylesheets) are used to generate automatically the translated schema. The tool works in three steps:

- In the first step, the original Akoma Ntoso schema is used with the *MapGenerator.xsl* stylesheet (see Appendix A), that creates the XML map file as shown before, but with no translated term (or, rather, with an identity translation from English to English). This tool automatically excludes non-translatable terms as specified in section 5.2.

- In the second step, an expert in the local language fills in the translated term for each item of the map. This file is then saved and made available to the next step.
- In the third step, the original Akoma Ntoso schema is used again, but with the *Map.xsl* stylesheet (see Appendix B) and the newly created map file. This stylesheet creates a valid XML Schema that is identical to the original Akoma Ntoso schema, but defines elements, attributes and enumerated values as specified in the mapping rather than as in the original.

The end result is an XML Schema with the correct namespace, all non-translatable terms unchanged, and all translatable terms translated wherever they appear in the original schema. No translation is performed of the name of complex types, simple types, element groups and attribute groups.

Appendix B: The MapGenerator.xsl stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsl:variable name="noTranslation" select='("a", "akomaNtoso", "b", "del", "div",
"i", "img", "ins", "li", "ol", "p", "span", "sub", "sup", "table", "td", "th", "tr", "u", "ul",
"alt", "border", "cellpadding", "cellspacing", "class", "colspan", "height", "href", "id",
"rowspan", "src", "title", "width" )'/>
    <xsl:template match="/">
        <akomaNtosoMap>
            <source      xml:lang="en"      namespace="http://www.akomantoso.org/20"
schemaLocation="akomantoso20.xsd" />
            <dest      xml:lang="en"      namespace="http://www.akomantoso.org/20/en"
schemaLocation="akomantoso20.xsd" />
            <xsl:variable                    name="elements"                    select="distinct-
values(//xsd:element/@name)" />
            <xsl:variable                    name="attributes"                  select="distinct-
values(//xsd:attribute/@name)" />
            <xsl:variable                 name="enumerations"                   select="distinct-
values(//xsd:enumeration/@value)" />
            <xsl:for-each select="$elements">
                <xsl:sort select="." />
                <xsl:if test="empty(index-of($noTranslation,.))">
                    <element source="{.}" dest="{.}" />
              </xsl:if>
            </xsl:for-each>
            <xsl:for-each select="$attributes">
                <xsl:sort select="." />
                <xsl:if test="empty(index-of($noTranslation,.))">
                <attribute source="{.}" dest="{.}" />
                </xsl:if>
            </xsl:for-each>
            <xsl:for-each select="$enumerations">
                <xsl:sort select="." />
                <xsl:if test="empty(index-of($noTranslation,.))">
                <enumeration source="{.}" dest="{.}" />
                </xsl:if>
            </xsl:for-each>
        </akomaNtosoMap>
    </xsl:template>
    <xsl:template match="*" />
</xsl:stylesheet>
```

Appendix C: The Map.xsl stylesheet

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:an="http://www.akomantoso.org/2.0">

   <xsl:preserve-space elements="xs:*"/>
    <xsl:output indent="yes"/>
    <xsl:variable name="map" select="document('./mappaitaliano.xml')/*"/>
    <xsl:template match="/">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="xs:element/@name|xs:element/@ref">
        <xsl:variable name="y"><xsl:value-of select="."/></xsl:variable>
        <xsl:variable name="x"><xsl:value-of
select="$map//element[@source=$y]/@dest"/></xsl:variable>
        <xsl:variable name="z"><xsl:value-of select="if ($x='') then $y else
$x"/></xsl:variable>
        <xsl:attribute name='{local-name()}'><xsl:value-of
select="$z"/></xsl:attribute>
    </xsl:template>

    <xsl:template match="xs:*|@*|*">
        <xsl:copy copy-namespaces="no">
            <xsl:apply-templates select="@*"/>
            <xsl:apply-templates xml:space="preserve"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="comment()">
        <xsl:comment>
            <xsl:value-of select="."/>
        </xsl:comment>
    </xsl:template>
        <xsl:template match="text()">
            <xsl:value-of select="."/>
    </xsl:template>

    </xsl:stylesheet>
```