Processing Hansard Documents with Portage

George Foster

November 12, 2010

1 Introduction

The House of Commons translation task involves handling documents that are transcriptions of parliamentary sessions (Hansard) or of the proceedings of various parliamentary committees. The documents are in XML, and generally contain parts where the source language is English, and parts where it is French. The desired output is a set of similarly-formatted XML documents containing both the original passages and their translations.

This task poses several problems for automatic processing with Portage: using specialized models for different sub-genres (eg Finance Committee proceedings versus Hansard), running translation in diffferent directions for different parts of a document, and preserving XML structure across translation. The last of these problems is particularly hard when XML markup occurs within a sentence, because it may be difficult or impossible to identify a corresponding marked-up segment in Portage's output.

This document describes the processing strategy that was used in the Translation Bureau trials of September 2010, and refined shortly afterwards. This includes the translation process only; it does not deal with model training, offline adaptation, or exploiting document structure to improve translation, for instance.

The processing strategy used for this task can be divided into parts that are Hansard specific, and parts that could be re-used in other settings. The rest of this document reflects this division: section 2 describes the overall strategy and how the two parts fit together; section 3 describes Hansard-specific steps, and section 4 describes general processing steps.

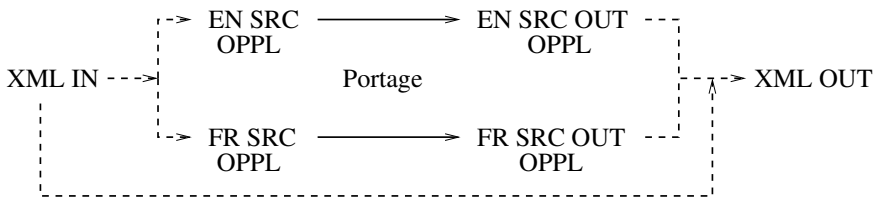

Figure 1: Overview of Hansard Processing

2 Overview

Figure 1 shows the complete processing strategy, with Hansard-specific steps indicated by dashed lines, and general steps by solid lines. The input XML document is first parsed to extract English-source and French-source segments, in one-paragraph-per line (OPPL) format, possibly containing intra-sentence XML markup. This is translated by Portage into a similar output format, which is then inserted into an output XML file, using the input file as a guide. Finally, the output XML file is split up and converted into a set of XML files in a slightly different final format.

All scripts for performing the Hansard-specific steps are in the directory `/home/portage/corpora/Hansard-HOC-2009/collect/scripts` (note that this directory also contains pre-processing scripts for training, which are not discussed here). All other scripts and programs are part of the normal Portage source code, mostly in the `preprocessing` module. The entire chain in figure 1 is carried out by the script `hanstrans.sh`, which calls the script `run-portage.sh` to perform the steps indicated with solid lines.

3 Hansard-Specific Processing

This section describes the script `hanstrans.sh` in general terms; for more specific help, do `hanstrans.sh -h`.

The input to `hanstrans.sh` is an XML file in "NRC" format, which essentially consists of a global header that identifies the session by date and kind (Hansard or specific committee), followed by a sequence of triples. Each triple consists of a header that contains information about the speaker, source language, etc., followed by three text segments—*Original*, *PreTranslated*, and *Translated*—which corresond to a transcription of an original speech, a partial translation for certain entities derived from a database, and the translation to be completed (initially empty). The text in the Original segment is usually a small number of paragraphs, roughly a speech or a segment of a speech by a single speaker. Each para-

```
<Original>
   <Intervention ToC="No"><Content><ParaText>It is a fundamental breach
   of the <Affiliation DbId="78738" Type="1">Prime Minister</Affiliation>'s
   duty to be accountable to the elected representatives of the Canadian
   people.</ParaText><ParaText>As the former House leader of my party,
   Stanley Knowles, is quoted as saying in the second edition of the
   <I>House of Commons Procedure and Practice</I>, on page 677, "Debate
   is not a sin, a mistake, an error or something to be put up with in
   parliament. Debate is the essence of parliament". I make this request
   in that spirit.</ParaText></Content></Intervention>
</Original>
<PreTranslated>
   <Intervention ToC="No"><Content><ParaText /></Content></Intervention>
</PreTranslated>
<Translated />
```

Figure 2: Example of a triple from an NRC format XML file (minus header information).

```
It is a fundamental breach of the <Affiliation DbId="78738" Type="1">Prime
Minister</Affiliation>'s duty to be accountable to the elected representatives
of the Canadian people.
As the former House leader of my party, Stanley Knowles, is quoted as saying in
the second edition of the <I>House of Commons Procedure and Practice</I>, on
page 677, "Debate is not a sin, a mistake, an error or something to be put up
with in parliament. Debate is the essence of parliament". I make this request
in that spirit.
```

Figure 3: OPPL text extracted from the XML triple in figure 2. Each line corresponds to a paragraph, and there are two (wrapped) lines.

graph is aligned across languages. Figure 2 shows an example.

Given this input, `hanstrans.sh` calls the script `extract-trans-from-hoc-xml.py` to extract the original paragraphs (ignoring pre-translations) and record their source language. The extraction procedure "flattens" and preserves selected sub-sentential markup as shown in figure 3. (Flattened elements are: *I*, *B*, *Sub*, *Affiliation*, *Document*, and *Query*.) The procedure also filters out `<Sup>` elements from French source segments. These are used to indicate superscript formatting, eg `146<Sup>e</Sup>`, which is not preserved in translation to English, eg `146th`. The result is two files, one English and one French, in OPPL format.

After extracting these two OPPL files, `hanstrans.sh` calls `run-portage.sh` to translate each direction concurrently, using genre-specific models that are se-

```
C'est une violation fondamentale du <Affiliation DbId="78738" Type="1">
premier ministre </Affiliation> sur son obligation de rendre des comptes
aux représentants élus de la population canadienne.
Comme l'ancien leader parlementaire de mon parti, Stanley Knowles, est
cité dans la deuxième édition de la <I> procédure et les usages de la
Chambre </I>, à la page 677, " le débat n'est ni un péché, ni une faute,
ni une erreur, ni quelque chose dont il faut s'au Parlement. Le débat est
l'essence même du Parlement ". Je fais cette demande dans cet esprit.
```

Figure 4: OPPL-format translation of the XML triple in figure 3. Each line corresponds to a paragraph, and there are two (wrapped) lines.

lected on the basis of the `Meta_OrganizationAcronym` attribute in the global XML header. The results are two output OPPL files with contents like the example in figure 4. Note that these files preserve sub-sentential markup from the input.

The next step is to insert the contents of the translated OPPL files back into the original XML document, within the appropriate `<Translated>` elements. This is done by the `replace-trans-in-hoc-xml.py` script, which follows essentially the same steps as `extract-trans-from-hoc-xml` in order to track the correct paragraphs in both OPPL files for insertion into the current `<Translated>` element. Text inserted into `<Translated>` elements is "deepened" so any intra-sentence XML structure will be correctly interpreted as such (rather than being escaped) when the final XML structure is written to the output file. A post-processing step also aims to re-insert `<Sup>` elements in French, eg `146 e` becomes `146<Sup>e</Sup>`. Figure 5 shows an example of the contents of the output NRC-format XML file from this stage.

The last step is to convert the output NRC-format XML file into a set of RTA (French: ATD) XML files. Each RTA file contains a unit of work destined for a translator, who is supposed to post-edit Portage's translations. In addition to the original and translated text, the RTA files contain formatting instructions for displaying the original and translated segments in an editor. The NRC to RTA conversion is performed by the `nrc2rta.pl` script, which uses the output NRC file to fill in the Translated segments in a set of input RTA files, writing one output RTA for each input RTA.

```
<Original>
   <Intervention ToC="No"><Content><ParaText>It is a fundamental breach
   of the <Affiliation DbId="78738" Type="1">Prime Minister</Affiliation>'s
   duty to be accountable to the elected representatives of the Canadian
   people.</ParaText><ParaText>As the former House leader of my party,
   Stanley Knowles, is quoted as saying in the second edition of the
   <I>House of Commons Procedure and Practice</I>, on page 677, "Debate
   is not a sin, a mistake, an error or something to be put up with in
   parliament. Debate is the essence of parliament". I make this request
   in that spirit.</ParaText></Content></Intervention>
</Original>
<PreTranslated>
   <Intervention ToC="No"><Content><ParaText /></Content></Intervention>
</PreTranslated>
<Translated><Intervention ToC="No"><Content><ParaText>C'est une violation
   fondamentale du <Affiliation DbId="78738" Type="1"> premier ministre
   </Affiliation> sur son obligation de rendre des comptes aux représentants
   élus de la population canadienne.</ParaText><ParaText>Comme l'ancien leader
   parlementaire de mon parti, Stanley Knowles, est cité dans la deuxième
   édition de la <I> procédure et les usages de la Chambre </I>, à la page 677,
   " le débat n'est ni un péché, ni une faute, ni une erreur, ni quelque chose
   dont il faut s'au Parlement. Le débat est l'essence même du Parlement ".
   Je fais cette demande dans cet esprit.</ParaText></Content></Intervention>
</Translated>
```

Figure 5: A triple in an output NRC XML file, corresponding to the example in figure 2.

4 Translating Paragraphs Containing XML

This section describes the `run-portage.sh` script in general terms; for more specific help, do `run-portage.sh -h`.

As described above, this stage is completely independent of Hansard. Input is OPPL raw text, possibly containing sub-sentential XML markup, as shown in figure 3; and output is also raw OPPL text, line-aligned to the input, with XML tags at the appropriate positions, as shown in figure 4. The problem of translating OPPL text with XML poses many difficulties for the standard Portage pipeline:

1. Paragraphs, rather than sentences, need to be preserved through translation.

2. The tokenizing and sentence-splitting procedure is not designed to work with embedded XML. Should XML tags be treated as separate tokens, or as if they weren't there, or as implicit whitespace?

3. The standard lowercase transformation is also not designed to work with XML. Ideally, this should not should not affect the contents of XML tags.

4. The decoder currently has an XML-like language for specifying translation rules; how should this interact with "foreign" XML?

5. The XML needs to be somehow passed through translation without affecting translation or language-model probabilities.

6. Ideally, source tokens that are bracketed by a tag pair should be translated into a contiguous sequence of target tokens. In general, for an arbitrary sequence of source tokens, there is no guarantee that this will happen: the corresponding target sequence may be discontinuous, or it may have boundaries that do not align with those of the source sequence (for example, if translation makes use of a phrase that spans the source sequence boundary).

7. The truecasing step is not designed to work with embedded XML. It needs to avoid modifying the case of text within texts, and also ignore tags when computing cased token-sequence probabilities.

8. The detokenizing step is not designed for XML either. Should tag pairs be placed adjacent to the tokens they bracket or separated with whitespace? Should punctuation that immediately precedes or follows a tagged sequence be made adjacent to the tags?

The following sections describe the solutions to these problems implemented in `run-portage.sh`.

4.1 Problem 1: Handling OPPL

Problem 1 is quite easy to solve. The tokenizer (`utokenize.pl`) currently optionally accepts OPPL text, and will enforce hard sentence boundaries at paragraph boundaries, and split sentences within paragraphs. The original paragraph boundaries can be remembered by inserting `<para>` tags on separate lines between the original paragraphs, eg:

```
First sentence. Second sentence.
<para>
Third sentence. Fourth sentence.
```

becomes:

```
First sentence .
Second sentence .
<para>
Third sentence .
Fourth sentence .
```

These tags form their own sentences, and go through the decoder without modification, so they can be used to convert output one-sentence-per-line (OSPL) text from the decoder into the desired OPPL format.

4.2 Problem 2: Tokenizing with XML

The solution to problem 2 should ideally depend on the nature of the XML markup, which in general can apply at, below, or above the level of tokens. However, since the goal is to pass it through translation intact, and since translation currently doesn't operate below the token level, we assume that markup *cannot* be incorporated within a single token, as in `extra<B>ordinary</B>`. To enforce this, a space character is added before and after each tag in a pre-processing step prior to tokenization. This prevents the tokenizer from concatenating a tag with any adjacent plain-text token. (Note that there is nothing that prevents sub-token markup from being used. It will just not be handled correctly, eg `extra<B>ordinary</B>` will turn into `extra <B> ordinary </B>`.)

The intention in separating tags with blanks is to have the resulting sequence of plain-text tokens be exactly the same as if all tags had been stripped and replaced by single space characters. In order to ensure this property, the tokenizer was modified to add XML tags to the list of tokens that are ignored for the purposes of full-stop detection, so that, eg:

```
... end of sentence. </tag> New sentence ...
```

gives:

```
... end of sentence . </tag>
New sentence ...
```

as desired, rather than leaving the original sequence as is. Notice that the tokenizer also correctly includes `</tag>` at the end of the first sentence.

The modifications to the tokenizer for handling XML markup are incomplete. Although there are no known violations of the property that tokenization decisions are identical whether tags are stripped and replaced with blanks or left in and separated with blanks, there are cases where introducing blanks interferes with correct decisions. Eg, in:

```
end of sentence. </x> ) Another sentence ...
```

`sentence.` is incorrectly treated as an abbreviation, and the sentence boundary is not detected. If `</x>` and the surrounding spaces are removed, this doesn't happen.

Another problem is markup that causes tokenization errors, such as the following, which occurs frequently in the Hansard:

```
<Affiliation...> President </Affiliation> 's initiative
```

Due to the space before `'s`, this is tokenized as:

```
<Affiliation...> President </Affiliation> ' s initiative
```

rather than retaining the original correct form. Fortunately, since this problem also occurs in the training corpus, the system learns how to fix it in most cases, and the resulting translation is not affected (although BLEU scores might be, due to incorrect tokenization in the reference).

4.3 Problem 3: Lowercasing XML

This was handled by adding a switch to `utf8_casemap` to tell it not to lowercase any text between < and > characters.

4.4 Problems 4, 5, and 6: Decoding with XML

The most thorough solution to the problem of preserving XML markup through translation would probably be to make the decoder aware of it. This would require extensions to `canoe`'s existing rule syntax, and modifications to the scoring procedure to make markup transparent to the LM, TM, etc. By itself, this would also not solve the problems of discontinuous translations or non-aligned boundaries mentioned above.

The solution implemented in `run-portage.sh` is much simpler: remove the XML markup before translation, and re-insert it afterwards, using phrase-alignment information to try to determine correct placement. This is accomplished by the separate program `markup_canoe_output`. In cases where the translation of a region bracketed by a pair of tags is continuous, and where phrase boundaries are aligned with the boundaries of the region, the placement of tags in the output will be perfect. When these conditions don't hold, `markup_canoe_output` relies on a set of heuristics to guess the best positions. These heuristics involve word-aligning phrase pairs using cognate information, bias toward the diagonal, and a small specialized dictionary and anti-dictionary currently heavily tuned for the Hansard (most likely harmless elsewhere). Although the heuristics work fairly well, the ideal approach would be to combine the use of `markup_canoe_output` with decoder rules like Moses' *zone* feature that force or strongly encourage XML-tagged regions of source text to translate as a unit (unfortunately these don't exist yet in Portage).

The requirement to strip then re-insert XML markup affects the standard Portage pipeline. Figure 6 shows what needs to happen, starting with an OSPL source file containing XML that has been tokenized and lowercased as described in the preceding sections (labelled `tok`, `lc`, `xml` in the figure). In general, this file will contain *two* kinds of XML markup: elements to be preserved through translation, as discussed above, and tags that are to be treated as part of the text, for instance the `<para>` markers required to handle OPPL.[1] The *strip xml* procedure removes the first kind of XML, but leaves in the second.[2]

The next step in the standard pipeline is to add rules for translating dates, numbers, etc. Although this is shown in figure 6 as taking lowercased, tokenized text as input, it is likely that it will need access to earlier stages in the pipleline (for instance, uppercase text) as well. It will also need access to the original file containing XML if it is to add zone-type rules to ensure that XML-tagged regions

[1] These could actually be treated either way, but they are a handy example, because `run-portage.sh` treats them as ordinary text tokens.

[2] In `run-portage.sh`, stripping is actually performed earlier in the pipeline for convenience, but the effect is the same.

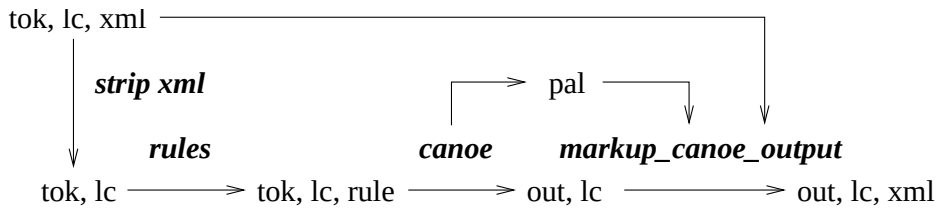

Figure 6: Pipeline for Handling XML during Decoding.

are translated contiguously and with phrase-aligned boundaries. Another function of this step is to escape XML that is to be treated as normal text, eg `<para>` ⇒ `\<para\>`. Currently this is all it does, as we do not yet have rules for English/French translation.

The next step in the pipeline is translation with `canoe`. This works as usual, except that it needs to generate an optional phrase-alignment file (labelled `pal` in the figure).

The final step is using `markup_canoe_output` to add XML from the original `tok,lc,xml` file to canoe's output. As described above, this uses information from the phrase alignment to determine the correct placement of tags in the output file. One subtlety is that it also needs to know which markup was left in the `tok,lc` file to be treated as ordinary text by canoe. This information is provided by specifying an explicit list of element names. For instance, if `para` is in this list, then all tags of the form `<para...>` or `</para>` will be treated as plain text tokens.[3] This solution is somewhat less flexible than `canoe`'s escaping procedure, but it should work in most cases. Note that ill-formed XML, or elements that span multiple sentences are ignored, which is perhaps not ideal behaviour.

4.5 Problem 7: Truecasing with XML

Truecasing poses a similar problem to decoding, in that having markup be ignored by the truecasing model would be difficult to implement.

As shown in figure 7, `run-portage.sh`'s strategy for truecasing is very similar to the strategy for decoding. First, the plain output from `canoe` (`out,lc` in the figure) is truecased as usual. Then `markup_canoe_output` inserts markup into this truecased file in the same places as in the lowercased version `out,lc,xml`, created in the previous step by transferring XML from the source file. The "pal" file used in this step just specifies a word-for-word alignment between `out,lc` and

[3] More precisely, all whitespace-delimited parts of these tags are treated as tokens.

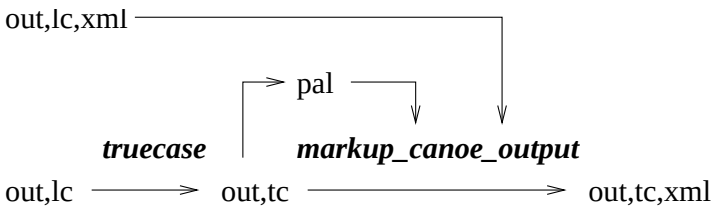

Figure 7: Pipeline for Truecasing XML.

`out,tc`, which of course leads to XML in exactly the same places in `out,lc,xml` and in the final output file `out,tc,xml`.

4.6 Problem 8: Detokenizing with XML.

Due to uncertainty about the desired conventions for detokenizing XML markup, the detokenization step was not modified for XML. In general, this means that tags (or whitespace-delimited parts of tags) are treated as ordinary tokens: they are concatenated with punctuation when called for, but otherwise left alone. For example:

```
( <x> some element </x> , ...
```

becomes:

```
(<x> some element </x>,
```